

Graded exercise

1) syscalls

1.) Wrappers.

The best the user space library kernel syscalls wrapper can do is check if $p == \text{NULL}$.

To check if the user space (Virtual memory) p given to it is even memory mapped thus paging to a physical memory address (through page tables). Work must be done on the kernel side. Same for read/write permission to that memory address.

Kernel check is mandatory. Library is if they want to do NULL check.

Graded exercise

1) Syscalls

1.1 Wrappers.

The best the user space library kernel syscall wrapper can do is check if $p == \text{NULL}$.

To check if the user space (virtual memory) p given to it is even memory mapped through paging to a physical memory address (through paging), work must be done on the kernel side.
Same for read/write permission to that memory address.

Kernel check is mandatory. Library is if they want to do NULL check.

1.2 fork

if $\text{fork} == 0$:
inside child

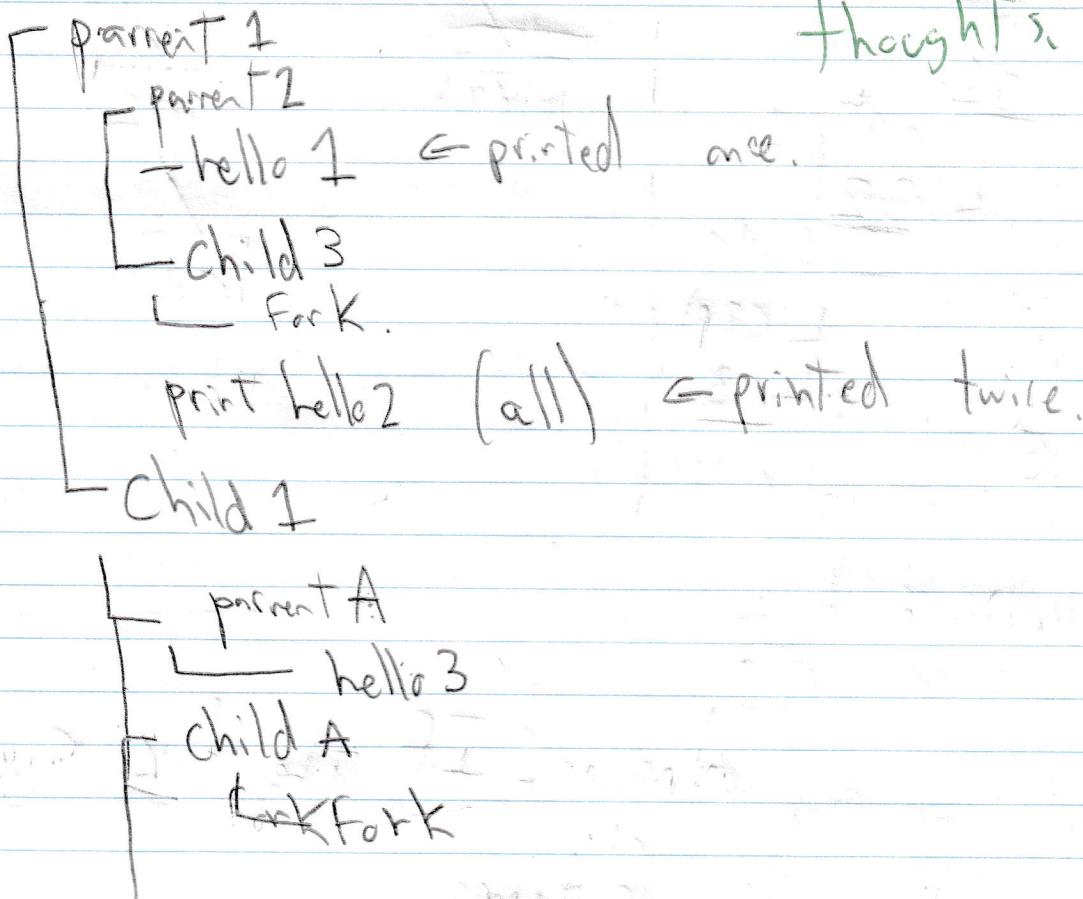
fork gives pid of child.
get ppid ↑, and get pid
parent pid exist.

if $\text{fork} != 0$:
inside parent.

pid_t = Typedef int

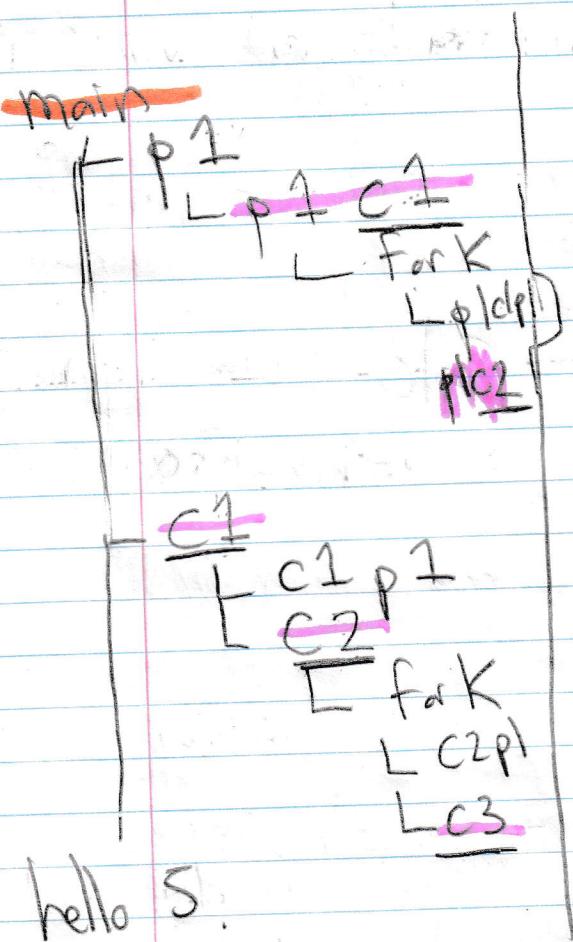
Assuming

calc and
thoughts



Ah, I don't need to check every print.

Since fork runs till the end of program.



Hello 5.

5 different child

1 parent

6 times.

6 times

Answer: If the fork succeeded,

2: Sync. in function

2.1

available resource is a global variable, it's available in all threads; is read and write (access and modify).

2.2

The part where available resource is written to / modified is by default a critical section which needs to be locked.

And here if available resource = const.

thread one pass the if statement but does not return, thread 2 does the whole function ~~and hence~~, then thread 1 return. Err.

So the if and return must be locked too.

hence, both function must be lock from start to finish.

2.2-2.3 (in between)

rewriting the function to allow
locking and unlocking before the return statement

wrapping the function call in lock, unlock in
the caller are possible solutions.

Lines with r&w access: 5, 8, 14

```
2.3 int dec (count) {  
    int ret;  
    lock;  
    if (avail) < count?  
        ret = -1  
    else  
        ret = 0
```

ret }

wnt) {

wnt;

(fines).

2.2-2.3 (in between)

rewriting the function to allow
locking and unlocking before the return statement

wrapping the function call in lock, unlock in
the caller are possible solutions.

Lines with r&w access: 5, 8, 14

2.3 int dec (count) {

 int ret;

 lock;

 if avail < count :

 ret = -1

 else

 ret = 0

 unlock;

 return ret }

int inc (count) {

 lock

 avail += count;

 unlock

 return 0; }

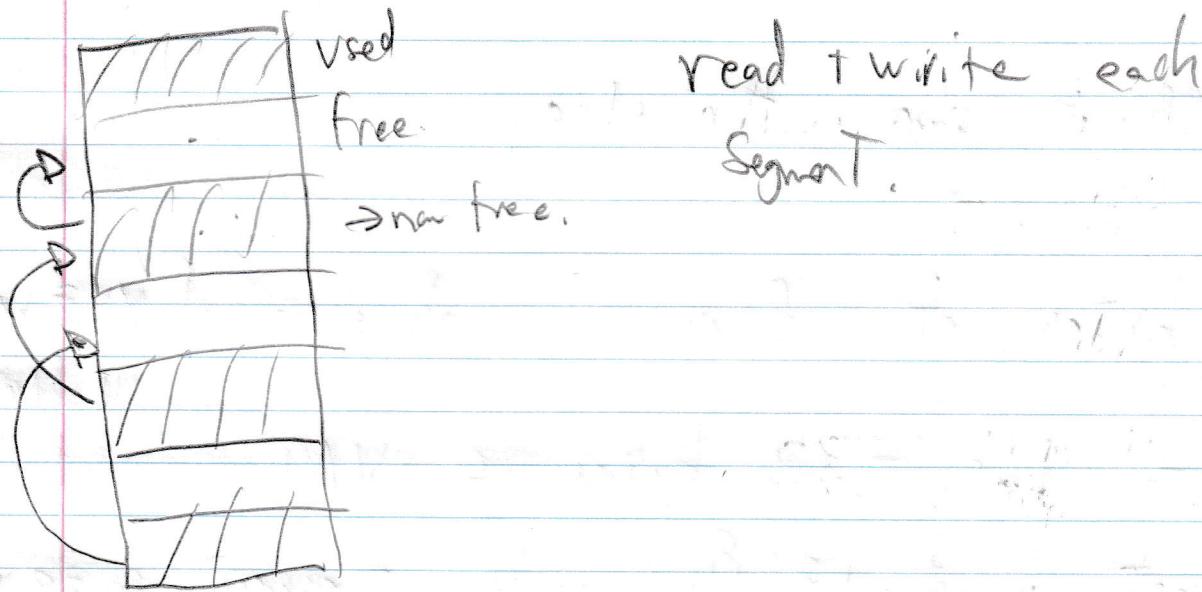
3. Compacting memory with buffers

3.1

$$4GB \approx \frac{1\text{ billion}}{2} = \frac{1e^9}{2} = 5e^8$$

$5e^8$ bytes, $\times (8 \text{ bytes}) \cdot \text{time}$

$$\text{read + write} = 4\text{ns} + 4\text{ns}$$



$$8\text{ns} \cdot 5e^8 = 4 \cdot 1e^9 \text{ ns} = 4\text{s}$$

if we need to make the whole thing, 4s.
else. f. 4s

↑ fraction of memory space.

4. Virtual mem, Segmentation

4.1

5 bit virtual address.

16 bytes of addressable phys mem.

$$2^{10} = 1024$$

$$2^5 = 32 \quad 2^6 = 64$$

64 KB of ram. (h No.)

16 bytes of ram. phys = dXYZW
↑
address

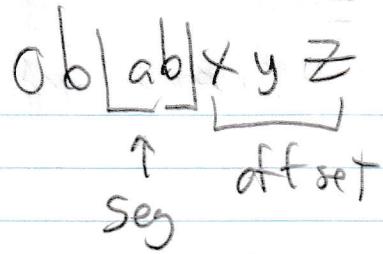
Program has 32 bytes of VM.

biggest band is 8. max segment size is 8

max segment address is $= 3 + \log_2 8$.

There's 4 segment: 2 bit for segment.

$2 + 3 = 5$, check. No $\frac{1}{2}$ or waste.
↑ ↑
Segment offset



$$V \Rightarrow p = \underset{\text{base}}{\text{Seg}[ab]} + \underset{\text{offset}}{xyz}$$

1. 00,000: \rightarrow 0b1100 W. (perm denied).

2. 11,001 \rightarrow 0b1000 + 001 = 0b1001 R
(pass)

3. 01,000 \rightarrow 0b0000 + 0: N/A (perm denied).

4. 10,010 \rightarrow 0b0001 + 010 = 0b0011 RW, pass.

5. 11,010 \rightarrow 1000 + 0,010 = 10010 R. (perm denied)

6. 00,111 \rightarrow 1100 + 0111} surpass add space. + 111 > bound
= 100
Seg fault.

7. 10,001 \rightarrow 0b0001 + 0,001 = 0b0010. RW, pass.

8. 00,000 \rightarrow 0b1100 W. pass.

9. 11,000 \rightarrow ob1000 + 0,100 = ob1000 R.
Segfault. (perm denied)

10. 10,100 \rightarrow ob0001 + 0,1000

100 > 011 = band Segfault.

1. Segfault

2. ob1001 \rightarrow 9

3. Segfault

4. ob0011 \rightarrow 3

5. Segfault

6. Segfault

7. ob0010 \rightarrow 2

8. ob1100 \rightarrow 12

9. Segfault

10. Segfault.

5. Virtual mem, Paging

5.1

Nice Answer encircled in
 $4KB = 12$ bit address. 2 page

1024 entry page dir $\Rightarrow 10$.

| Dir Index | Page Index | Offset |
|-----------|------------|--------|
| 10 | 10 | 12 |

Page directory = 1 array.

& page directory = CR3.

Dir $IdxA = CR3 + Dir\ Idx \cdot 4$

32 bits is 4 bytes

$1024 \cdot 4$ bytes
= 4 KB page
dir.

PTBA = * Dir $IdxA$,
Page Table base address.

Table $IdxA = PTBA + Table\ Idx \cdot 4$

1024 entry.

4 KB = 4 MB

For All page
Tables.

PFBA = * Table $IdxA$

Page Frame Base Index.

Phys Address = PFBA + Page Offset.

4.064 MB, for the whole thing if it's fully allocated

5.1 answer: 1024, See Next

Even more steps for myself. page for

As shown before, a page table has

2^{10} entry, so 1024.

and since they contain address of 4B size
then its memory size is 4KB

If one page frame has 4KB of phys mem,

It has $\frac{4096}{\text{Arch bits (32)}} = 1024$ 32 bit words
byte size (1) however...

$1024 = 2^{10}$. However, we can check the bytes, not just words, as char is a type and there's byte size registers.

So, offset = \log_2 (Page frame size
in bytes)

$$= \log_2 (4096) = 12.$$

The dir has 1024 entries, so 10

$$32 = 10 + x + 12$$

$$x = 10.$$

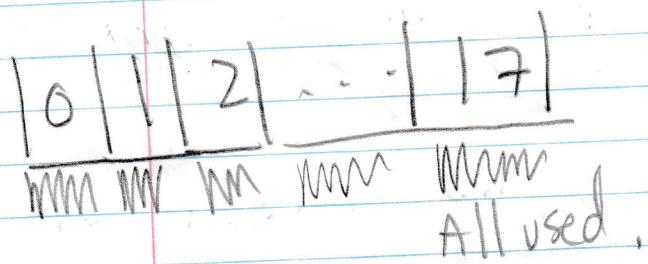
10 bits are for the page table entry.

$$2^{10} = 1024.$$

A page table has 1024 entries, and takes 4KB of memory,
which by luck is the same as page frame size

5.2 Spatial locality

Assuming $[0, 17] = \underline{0, 1, 2, \dots, 17}$



So 18 MB is used. $\frac{18 \cdot 1024}{4} = 4608$ = 4068

Assuming $[90, 117]$ $\frac{1024}{4} = 256$

So $117 - 90 = 27$

$27 \cdot 256 = 6912$

$4608 + 6912 = 11520$ page frames
used.

$\frac{11520 \text{ page frames}}{1024 \text{ page frame}} = 11.25 \text{ page tables}$

1 dir entry does $1024 \cdot 4\text{KB} = 4\text{MB}$

10 10 12 but virtual memory contiguous

$\frac{18}{4} = 4.5$, so not fully aligned,

half of the 5th page table will be filled.

$\frac{90}{4} = 22.5$, $\frac{117}{4} = 29.25$.

but, virtual page tables can

have holes, so it doesn't matter.

You can have the first 512 (half) page tables be null / unallocated

It doesn't need to have an entire page table nor. Up to down. So the 75% from 117 can be unused.

5.3 , $64 = 2^6$ checks at

6-bit V \rightarrow 6bit Phys.

Page frame is 4 byte, 4 entry.

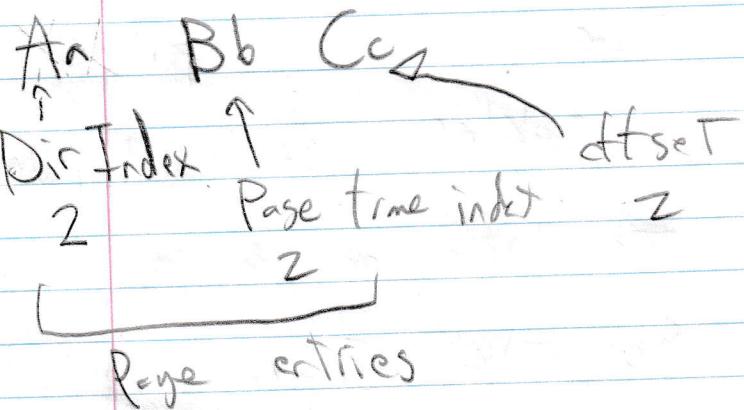
Page size = page frame size
[] []
Virtual Physical

so the offset match.

$\log_2(4) = 2$. offset is 2 bit.

Dir size = $\log_2(4 \text{ entry})$
6-bit

Page size = $\log_2(4) = 2$
bit



Page dir is at frame 2 (#3)

Page dir:

8 → invalid
9 → 0
10 → 5
11 → 13

Page Table 0

0 → 0xd1
1 → 0xf7
2 → 0xbf
3 → 0x00

4.064 MB for the whole thing if it's fully allocated

5.1 answer: 1024 See Next
Even more steps for myself. page for

As shown before, a page table has

2^{10} entry, so 1024.

and since they contain address of 4B size
then its memory size is 4KB

If one page frame has 4KB of phys mem,

It has $\frac{4096}{\text{Arch bits (32)}}$ = 1024 32 bit words
byte size (1) however....

$1024 = 2^{10}$. However, we can check the bytes, not just words, as "char" is a type and there's byte size registers.

1) $00,00,00 \rightarrow$ invalid $5,3,3$
invalid

2.) $11,00,10 \rightarrow$ invalid
3 ↑
invalid

3) $10,11,01 \rightarrow 13,11 = + - (= 1110)$
↑ ↑ ↑
base2 {3 offset

4). $01,01,11 \rightarrow 7,3,3 = = a = 110$
↓
7

Page 0: invalid

5.3, 2

Page 1:

C0 \rightarrow 7

C1 \rightarrow 7

C2 \rightarrow 15

C3 \rightarrow invalid.

Page 2

C0 \rightarrow ~~10~~ 11

C1 \rightarrow invalid.

C2 \rightarrow 12 (c)

C3 \rightarrow 13 (d)

Page 3 (Frame 3)

C0 \rightarrow invalid

C1 \rightarrow 10 (a)

C2 \rightarrow 4

C3 \rightarrow 8

5.3) 2

Page dr:

0 → invalid

1 → frame 0

2 → frame 5

3 → frame 3

00 → invalid

01 → frame 000

10 → frame 101 (5)

11 → frame 011 (3)
