Graded exercise

1) Syscalls

1.1 Wrappers.

The best the user space Library kernel sysca
wrapper can do is check if p == NULL.

To check if the user space (virtual memory)
p given to it is even memory mapped thro
paging to a physical memory address (through pag
work must be done on the kernel side.
Same for read/write permission to that me
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 == 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 fork==0:
inside child

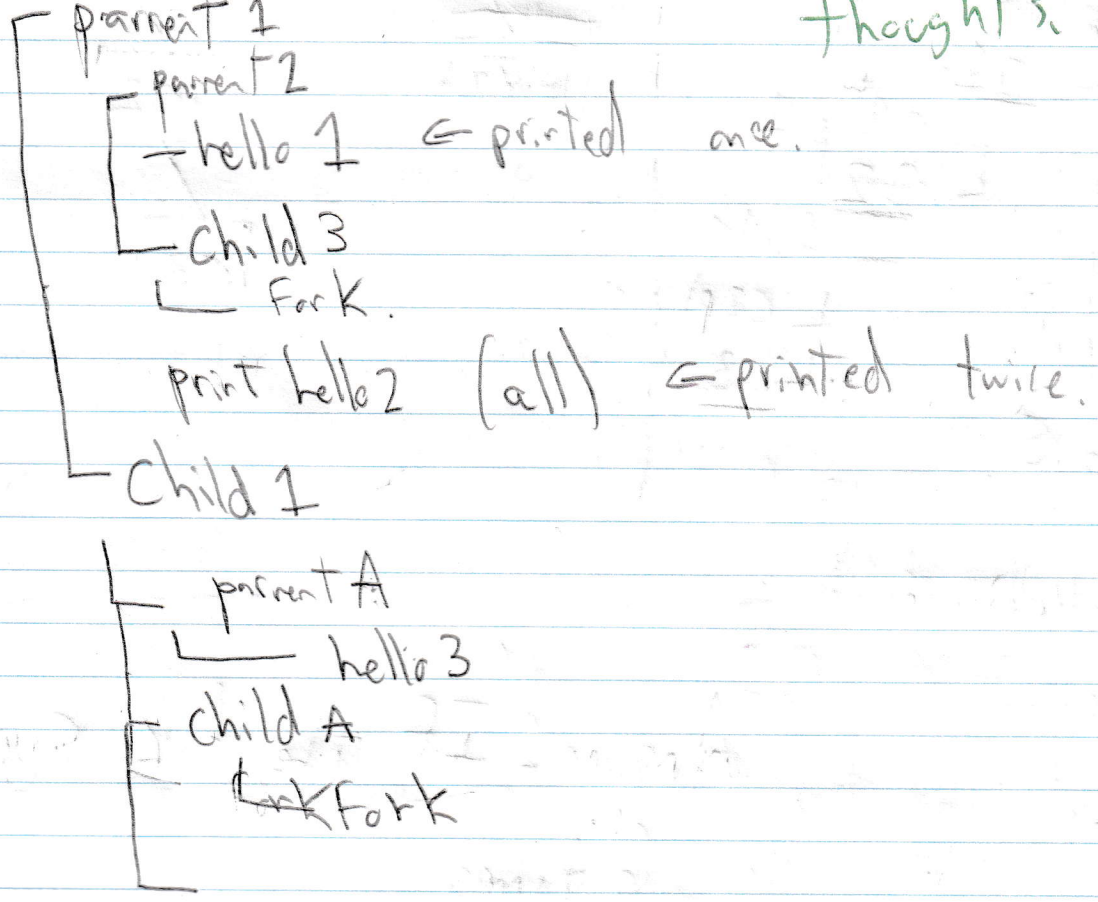
if fork!=0:
inside parent.

Assuming

fork gives pid of child.
getppid, and getpid
↑ parent pid exist.

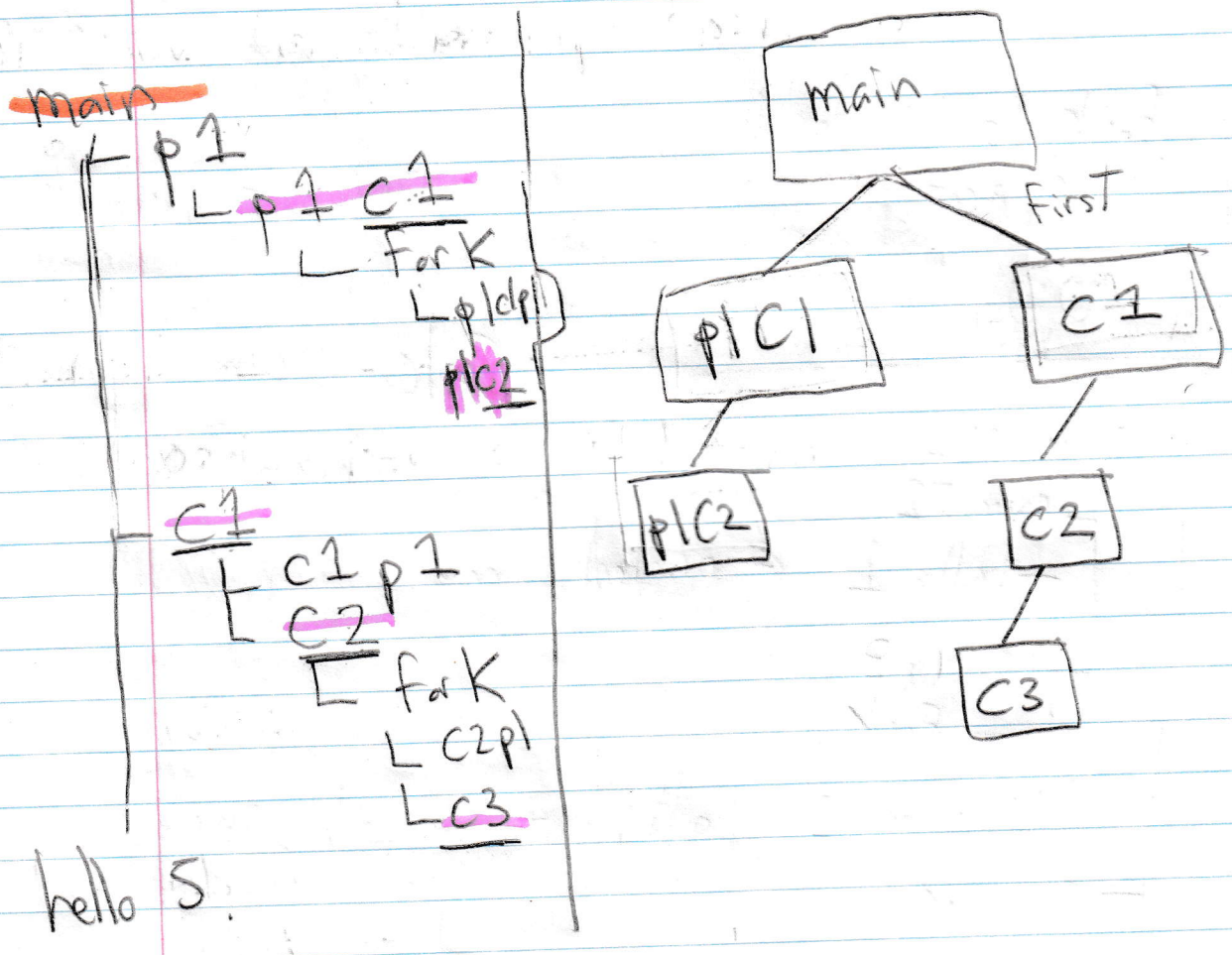
pid_t = Typedef int

Calc and thoughts.



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

Since fork runs till the end of program.



5 different child

1 parent.

6 times.

Answer: If the fork succeeded,

6 times

2: Sync.

2.1

available ressource is a global variable, it's available in all threads, is read and wrote. (access and modify).

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

And here if available ressource = count.

thread one pass the if statement but does not return, thread 2 does the whole function ~~and close~~ ~~then~~, then thread 1 return: Error.

So the if and return must be locked too.

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

2.2-2.3 (in between)

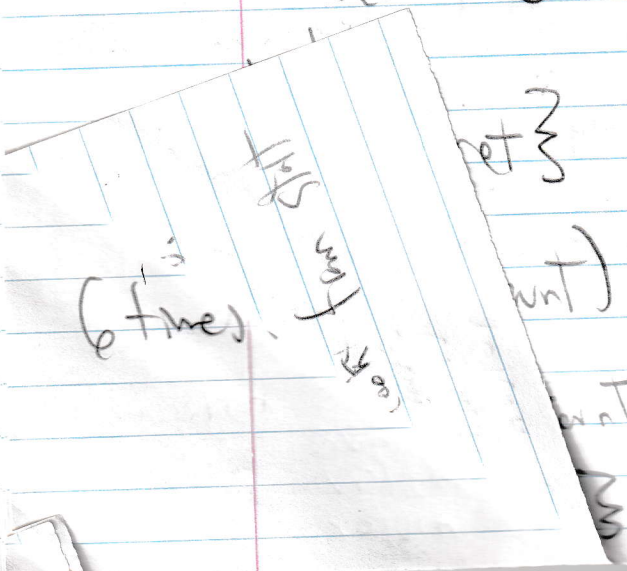
rewritting the function to allow locking and unlocking before the return statement

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 < cnt;
    ret = -1
else
    ret = 0
```



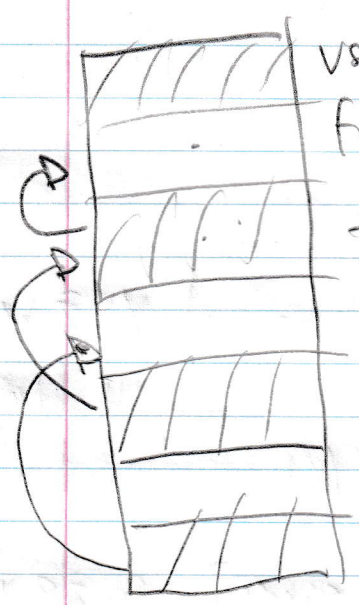
3. Compacting

3.1

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

$5e^8$ bytes $\cdot$ (8 bytes) $\cdot$ time

read + write = 4ns + 4ns.



read + write each segment.

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

if we need to move the whole thing, 4s.

else. f. 4s

↑ fraction of movement space.

4. Virtual mem, Segmentation.

4.1

5 bit virtual address.

16 bytes of addressable phys mem.

$2^{10} = 1024$

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

64KB of ram. Ah No,

---

16 bytes of Ram. , phys = ab XYZW
↑4 address

Program has 32 bytes of VM.

biggest band is 8. max segment size is 8

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

There's 4 segment: 2 bit for segment.

2 + 3 = 5, check. No waste.

↑ ↑
segment offset

0b | ab | x y z

↑ seg offset

V → p = Seg[ab]$_{base}$ + xyz

1. 00,000 → 0b1100 W. (perm denied).
2. 11,001 → 0b1000 + 001 = 0b1001 R (pass)

3. 01,000 → 0b0000 + 0 : Naive (perm denied).
4. 10,010 → 0b0001 + 0010 = 0b0011 RW, pass.
5. 11,010 → 1000 + 0,010 = 1010 R. (perm denied)

---

6. 00, 111 → 1100 + 0111; surpass add space. + 111 > bound = 100
   Seg fault.
7. 10,001 → 0b0001 + 0,001 = 0b0010. RW. pass.
8. 00,000 → 0b1100 W pass.

9. 11,000 → 0b1000 + 0,1100 = 0b10100 R.
Segfault. (perm denied)

10. 10,1100 → 0b0001 + 0,1000
100 > 011 = bound. Segfault.

---

1. Segfault
2. 0b1001 → 9
3. Segfault
4. 0b0011 → 3
5. Segfault
6. Segfault
7. 0b0010 → 2
8. 0b1100 → 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 ⇒ 10.

Dir Index	Page Index	offset
10	10	12.

Page directory = 1 array.

&page_directory = CR3.

DirIdxA = CR3 + DirIdx · 4

PTBA = *DirIdxA.
Page table base address.

TableIdxA = PTBA + TableIdx · 4

PFBA = * TableIdxA
Page Frame Base Index.

Phys Address = PFBA + Page Offset.

32 bytes is 4bg/g

1024. 4 bytes
= 4 KB page dir.

1024 entry.
4KB = 4MB
For All page tables.

4.004 MB for the whole thing if fully allocated

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

As shown before, a page table has 2^{10} entry, so 1024.

and since they contain address of 4B size then it's memory size is 4KB

If one page frame has 4KB of phys mem,

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

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 $[0, 18[= 0, 1, 2, \dots, 17$

0	1	2	...	17

All used.

So 18 MB is used. $\frac{18 \cdot 1024}{4} = 9 \cdot 512 = 4608$

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

So $117 - 90 = 27$

$27 \cdot 256 = 6912$

$4608 + 6912 = 11520$ page frames used.

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

- 1 dir entry does $1024 \cdot 4\,KB = 4\,MB$

10 10 12 but virtual memory contiguous

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

half of the 5th page table will be filled.

- $\frac{90}{4} = 22.5$, $\frac{117}{4} = 29.27$.

but, virtual page tables can have holes, so it doesn't matter.

you can have the 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 out

6 bit V → 6 bit Phys.

Page frame is 4 byte, 4 entry.

Page size = page frame size

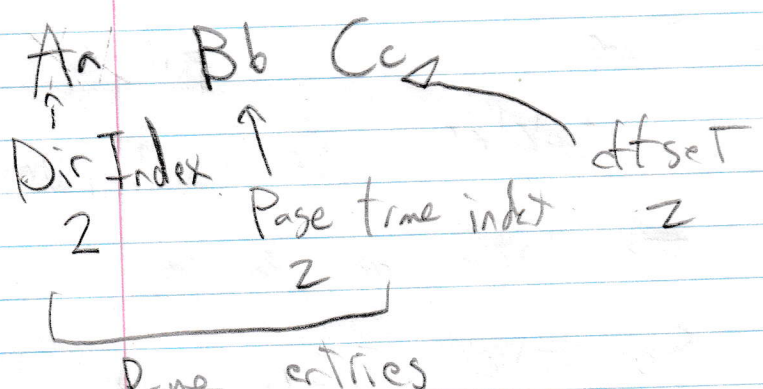
virtual physical

so the offset match.

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

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

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



Page dir is at frame 2 (# 3)

Page dir:

8 → invalid
9 → 0
10 → 5
11 → 13

Page table 0

0 → 0xd1
7 → 0xf7
2 → 0x6f
3 → 0xc0

4.004 MB for the whole thing if fully allocated

5.1 answer: 1024, See Next page for ☐

Even more steps for myself.

As shown before, a page table has 2^{10} entry, so 1024.

and since they contain address of 4B size then it's memory size is 4KB

If one page frame has 4KB of phys mem,

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

however....

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

2.2-2.3 (in between)

rewritting 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 < Cont;
    ret = -1
else
    ret = 0
vnlock;
return ret}

int inc (count) {
  lock
  avail += Count;
  Unlock
  return 0; }
```

1) 00, 00, CC → invalid (invalid) 5.3, 3

2.) 11, 00, 10 → invalid
3 ↑ invalid

3) 10, 11, 01 → 13, 11 = [illegible]
base2 13 offset
d

4). 01, 01, 11 → 7, 3 3 = [illegible]
7

5.3, 2

Page 0: invalid

Page 1:

00 → 7

01 → 7

02 → 15

03 → invalid.

Page 2

00 → [illegible]

01 → invalid.

02 → 12 (c)

03 → 13 (d)

Page 3 (Frame 3)

00 → invalid

01 → 10 (a)

02 → 4

03 → 8

5.3, 7

Pagedir:

0 → invalid
1 → frame 0
2 → frame 5
3 → frame 3

00 → invalid
01 → frame 000
10 → frame 101 (5)
11 → frame 011 (3)