COSC 420 - High-Performance Computing
Project 1

Dr. Joe Anderson

Due: 17 October

1 Preliminaries

In modern Linux systems, the `crypt` library (see `info crypt` or the man page, `man 3 crypt` or http://man7.org/linux/man-pages/man3/crypt.3.html for some details) to encrypt and store user passwords. This data is stored typically in the file named `/etc/shadow` and is only readable by the system root. Different systems use various encryption algorithms, and when using the `glibc` version (which we are on the COSC server), you can additionally specify which algorithms are employed. As a technical note, you will have to have the pre-compiler macro `#define _XOPEN_SOURCE` above your `#include` directives, as well as compile with the `-lcrypt` linker argument.

In this project, you will play the role of a "hacker" (possibly hired by the company to do a test of their security) who has obtained access to a system and made a copy of the shadow file. Back on your own system, you aim to find out which users have chosen an unsecure password, using what is known as a "dictionary attack"; in short, you will try and guess which password is correct, by brute force.

In the shadow file you will find user data in the following format (see `man 5 crypt` for information on this format:

```
<username>:$<algorithm id>$<salt string>$<encrypted password>
```

for example

```
jtanderson:$1$gA$Po1AuQSRCorWXHi8cIOhK/
```

On this line, the `<username>` segment is the login name of the user. The `<algorithm id>` is an integer, usually 1, 5, or 6 to indicate which hashing algorithm was used to encrypt the password string (see the `crypt` documentation for a list of which algorithms use which id). The `<salt string>` is a string of nonsense characters that is appended to the user's password in order to obfuscate the encrypted strings against hackers like you; how this works is that, if there were no salt, you could pre-encrypt all the possible passwords beforehand, and simply look up the encrypted version in your pre-computed database. Because the salt is appended to the plaintext password, this means your pre-computed dictionary encryption is invalid! For example, if the salt is "abc" then when a user's password is "fluffybunny", what is *actually* encrypted is "abcfluffybunny". The `<encrypted password>` section is *part of* the output of the `crypt` function, when passed the salt, algorithm id, and unencrypted password string. The full output is `$id$salt$cypertext`.

The sequential version of your brute force password-guessing algorithm will go as follows:

To parallelize this: you have some choices. You can split the candidate words across each processor, the prefixes, the suffixes, and even the individual passwords to be brute-forced. Think carefully about the pros/cons of each of these, and whether or not you can even combine them!

2 Objectives

In this project you will focus on the following objectives:

```
1: for Every line in the shadow file do
2:     Parse the line into username, id, salt, and cyphertext
3:     for Every word, w, in the dictionary do
4:         for Every numerical prefix p and suffix s of integer strings (including empty string) do
5:             Let guess := ws or pw depending on if you are testing a prefix or suffix
6:             Calculate result := crypt(guess, "$id$salt$")
7:             Compare result to the second field of the parsed line. If they match, you've cracked it!
8:         end for
9:     end for
10: end for
```

1. Develop familiarity with `c` programming language
2. Develop familiarity with parallel computing tools `MPICH` and `OpenMPI`
3. Develop familiarity with parallel file I/O

3 Tasks

1. You may work in groups of one or two to complete this project. Be sure to document in comments and your `README` who the group members are.
2. You will read in the dummy `/etc/shadow` file and use MPI to try brute forcing users' passwords.
   (a) Make sure you download the dictionary file (words.txt) from the course webpage
      i. Fun fact: this is just a copy of a standard `/usr/share/dict/words` file used on GNU/Linux for spell checkers!
   (b) Download the sample `shadow` file from the course webpage: shadow.
3. Use MPI and use multiple nodes to find which users have passwords of dictionary words, perhaps followed or preceeded by short strings of integers, e.g. "99balloon", "1337dude", or "password123".
   (a) You may assume that neither the prefix nor suffix integer does not exceed 4 characters.
   (b) You may assume that each dictionary password has a prefix *or* a suffix number, but not both.
   (c) You may assume standard grammatical capitalization, i.e. proper nouns start with a capital. Note that the dictionary file *does* include many proper nouns like common names!
   (d) You may assume that no special characters appear in any password.
   (e) You may hard-code the number of lines in the `words` file, once determined. You can check this with the command-line utility `wc -l words` to count the lines. This will save you having to loop the file and count manually in the program.
   (f) **Not all users in the system use this strict dictionary word scheme!** But you must find out which do and which do not.
4. Use a **shared** output file where the cracked passwords can be stored in a user-readable format.
5. Make sure you use MPI's shared file access functionality to define "windows" from which each node can access the output files – you can hard-limit each line to be 255 characters long and pad out any extras with spaces. If a particular password cannot be cracked, you can report a notice as such.
   (a) Created some derived `MPI_Datatype`'s will likely be a good idea to simplify logic

(b) You may want some message-passing to "abort" (but not actually `MPI_Abort`) the brute forcing of specific passwords, once one process has cracked it – non-blocking messages may be helpful here.

6. In addition to storing the results in an output file, communicate the hacked passwords back to the root to be printed to standard output.

7. Note that for debugging output, it may be helpful to have each node write debug output to a file unique to that process. This will be how we will gather logs when submitting to larger clusters.

8. Use standard timing tools (c `time_t` or `MPI_Wtime`) to record the time it takes to complete your password crack. Compare using different numbers of cores (as practical): e.g., 10, 20, 50, 100

9. Include a `README` file to document your code, any interesting design choices you made, and answer the following questions:

   (a) What is the theoretical time complexity of your algorithms (best and worst case), in terms of the input size?

   (b) According to the data, does adding more nodes perfectly divide the time taken by the program?

   (c) Consider the problem of brute-forcing passwords under *only* maximum string length. How much time would it take to complete this hack, extrapolating from your measurements?

   (d) What are some real-world software examples that would need the above routines? Why? Would they benefit greatly from using your distributed code?

   (e) How could the code be improved in terms of usability, efficiency, and robustness?

4 Submission

All submitted projects must compile with `mpicc` and run on the COSC Linux environment. Include a `Makefile` to build your code. Upload your project files to MyClasses in a single `.zip` file. Finally, turn in (stapled) printouts of your source code, properly commented and formatted with your name, course number, and complete description of the code and constituent subroutines. Also turn in printouts reflecting several different runs of your program (you can copy/past from the terminal output window). Be sure to test different situations, show how the program handles erroneous input and different edge cases.

5 Bonus

(10 pts each) There are several users who have made their passwords *slightly* more secure, by using multiple words, character substitutions, and special characters. Each one of these that you crack will be worth extra points!

① All Nodes
- Read shadow file and store on rocessub

② All Nodes
- Read in a portion of the dictionary size / worldsize

③ All Nodes
• try to find password for user1 in shadow file.

psuedo for step 3. below ↓

④ All Nodes
- parse string into individual words look at end of line char '\0' then send to function → strtok (string, ' ')

MAIN:

- every node Reads part of the file. OR
- Root Reads chunk and scatters

for (i=0 to Node's dict portion size of) {

//Parse word from string
look at NULL char

if (checkword(pswd, dictBuff[i]) == TRUE)

1. write password to finished file, with username associated with it.
2. communicate to all Nodes to stop trying to work to crack the current user's pswd. Move on to next usrs.

check word function pseudo code

from shadow.txt words.txt

```
bool checkWord(pswd, word)

    for i = 0  <=  9999
        prefix = i + word
        suffix = word + i
    if (HASH(prefix) == pswd)
        return TRUE
    if (HASH(suffix == pswd)
        return TRUE

return false; // user password was not found testing the word from the dictionary
```

testing for standard pswd format

would it be more efficient to test every users pswd since we generated the hash already?

moving the 1000, 100, 10, 1

Reading char ** buffer

1.

2. allocate array of strings
   char *[# strings] // does this need to be malloced by # of lines
   = malloc(dict size/world)

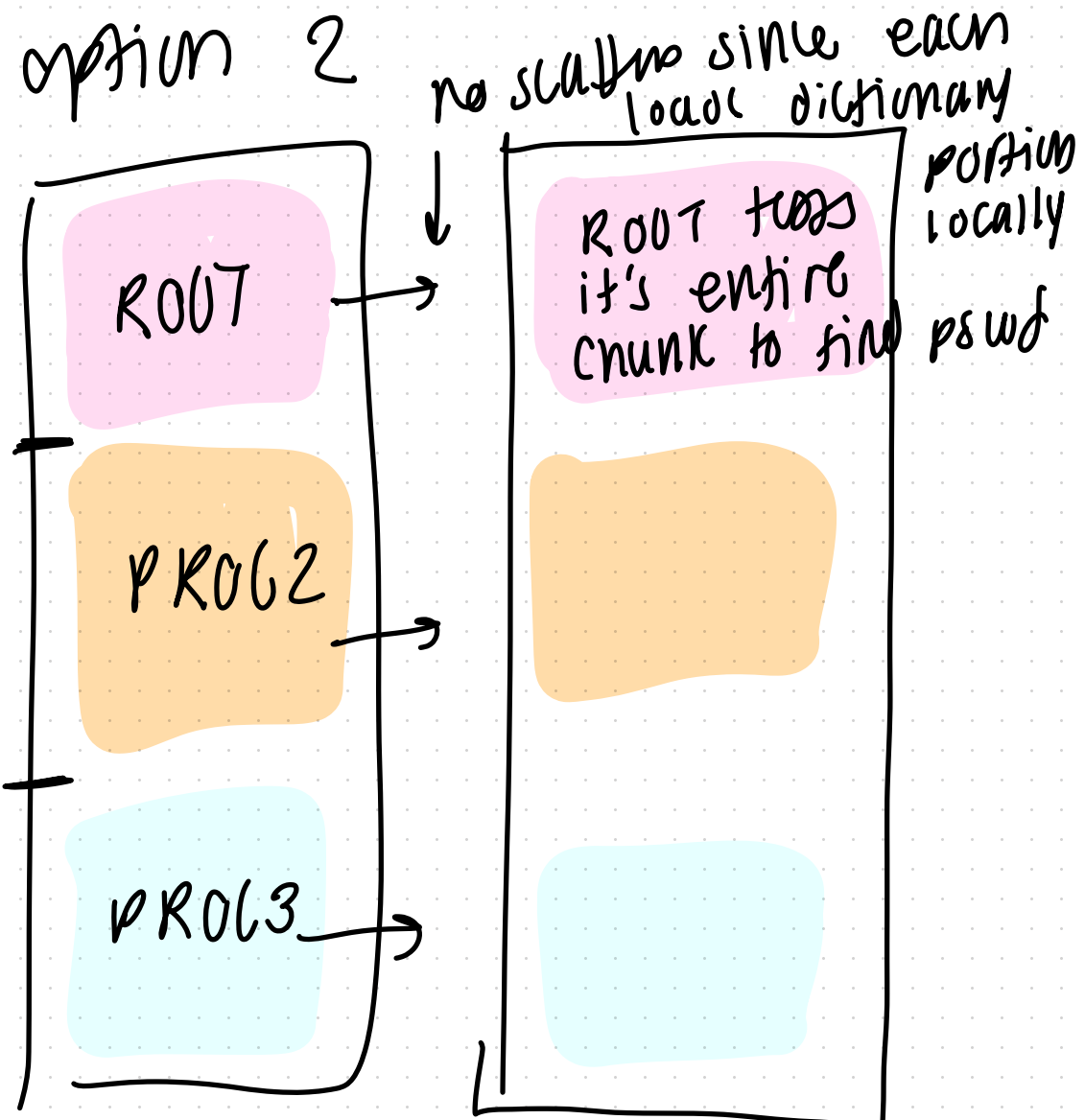
3. read in string w/ fgets? then store into main string

   char * :
   memset to '\0'
   sprintf into char* and move forward 255 char every time.

TODO: Nodes - boxes vlum

1. How to SBATCH on linux
   Nodes → sinfo
2. test to ensure all words from dictionary are being read in
3. how to communicate btwn Nodes new to step working ie, we found the users password
4. structuring the shared file that we found the pswds in
5. save pointing
   - every 200 lines

option 2

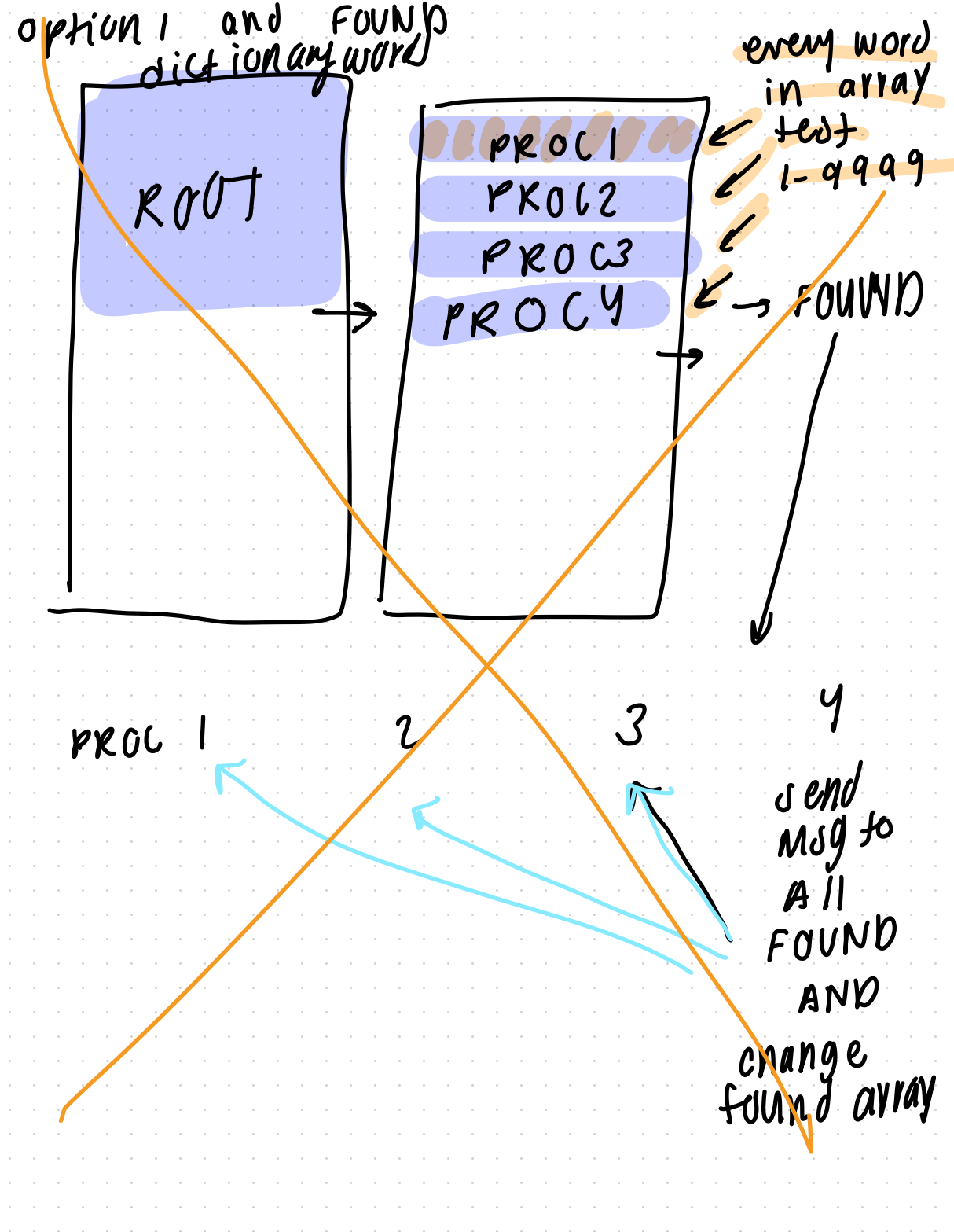


Every Node reads in a portion of dict which gets stored locally.
Every Node begins testing for pswd
If found send to all Nodes

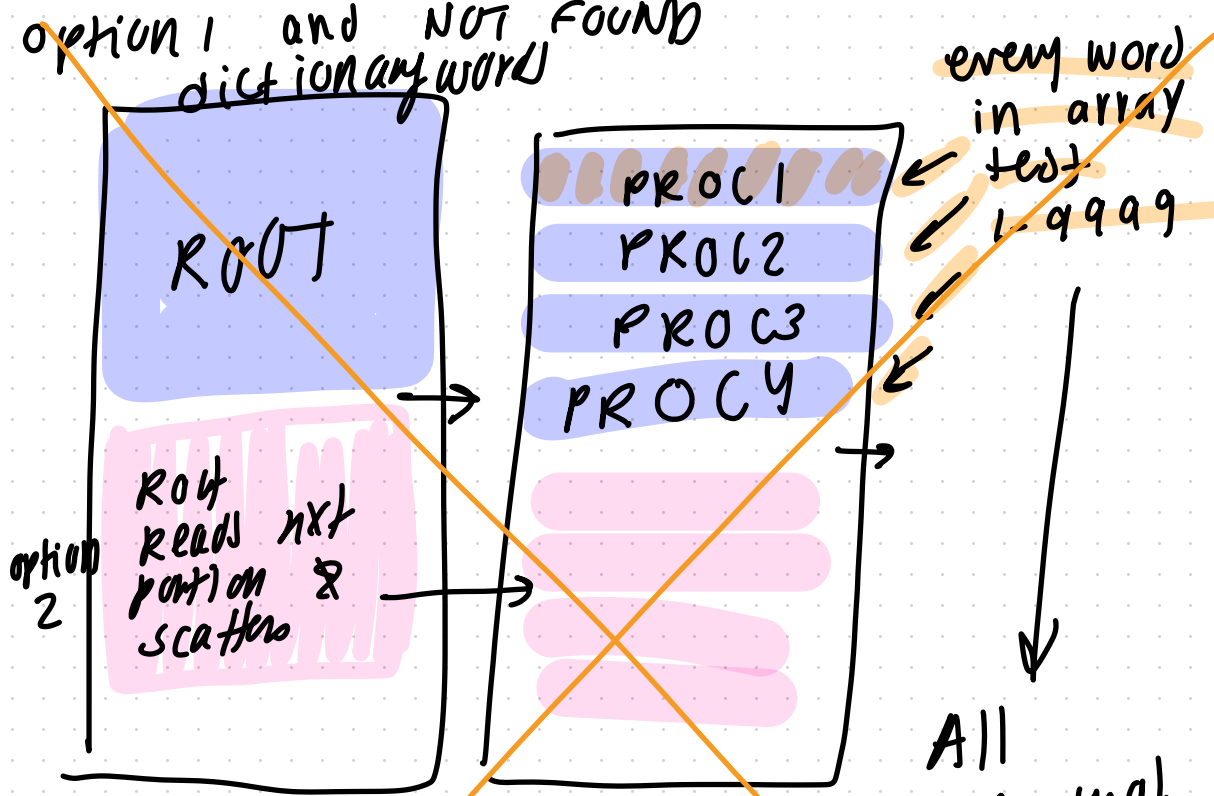
to step working to find that current pswd. update found array, everyone continues to find next pswd. if none found send msg then we know that pswd is of a special format.

- might not be able to do if file is too long

the rest is less organized


option 1 and FOUND
dictionary word
ROOT
PROC1
PROC2
PROC3
PROCY
every word
in array
test
1-9999
FOUND
PROC 1
2
3
4
send
Msg to
All
FOUND
AND
change
found array

- scatterv
- send the line # (after Math) for each processor to begin reading file from.



option 1:
- w/ all this loaded into memory we can check user 2 password.

None have found the users password.

option 2:
- we can load in 2nd dictionary and continue to try and find 1st repeat prev

I think we should do it where we are more focused on using, then throwing away pts of dictionary, vs. trying to find every users pswd. sequentially.

Chloe Notes

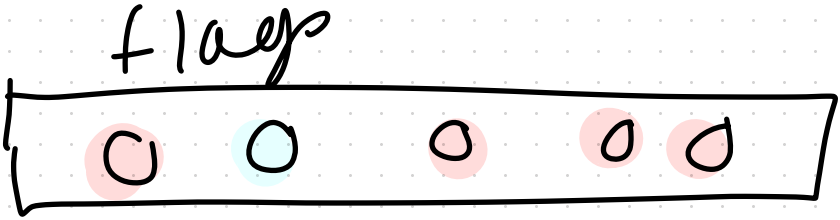
- test w/ something we already knew, generate a main and textfile to test.

- could use array of int
  if 0 → NOT found
  1 → FOUND
  shared array.

- get rid of file I/O and string operations

- brute force
  - list of flags, and ignore if the flags are done or not. every loop, iterating & check if user is done.

for chunk of dict
  - check if hash == any name in list



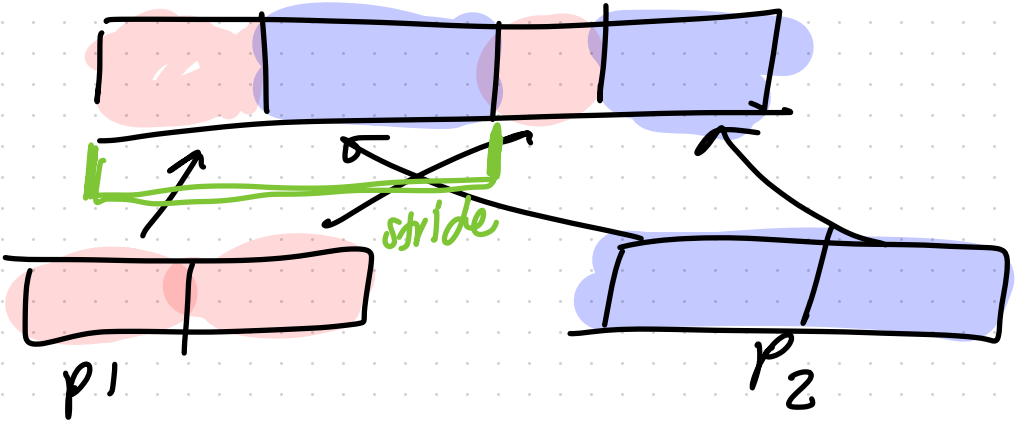
do I have a message? probe

- give everyone every password, split dictionary.

build in save pointing
- record in file
what word you left
off on to save
time. Record metadata
don't use too much
because file
I/O is costly.

- to think about what
node did where in the
file use cursor
memory.

window unfe.cpp



To write to a file

1. make a type datatype
2. make another type to create the extent
3. → MPI_File set file view

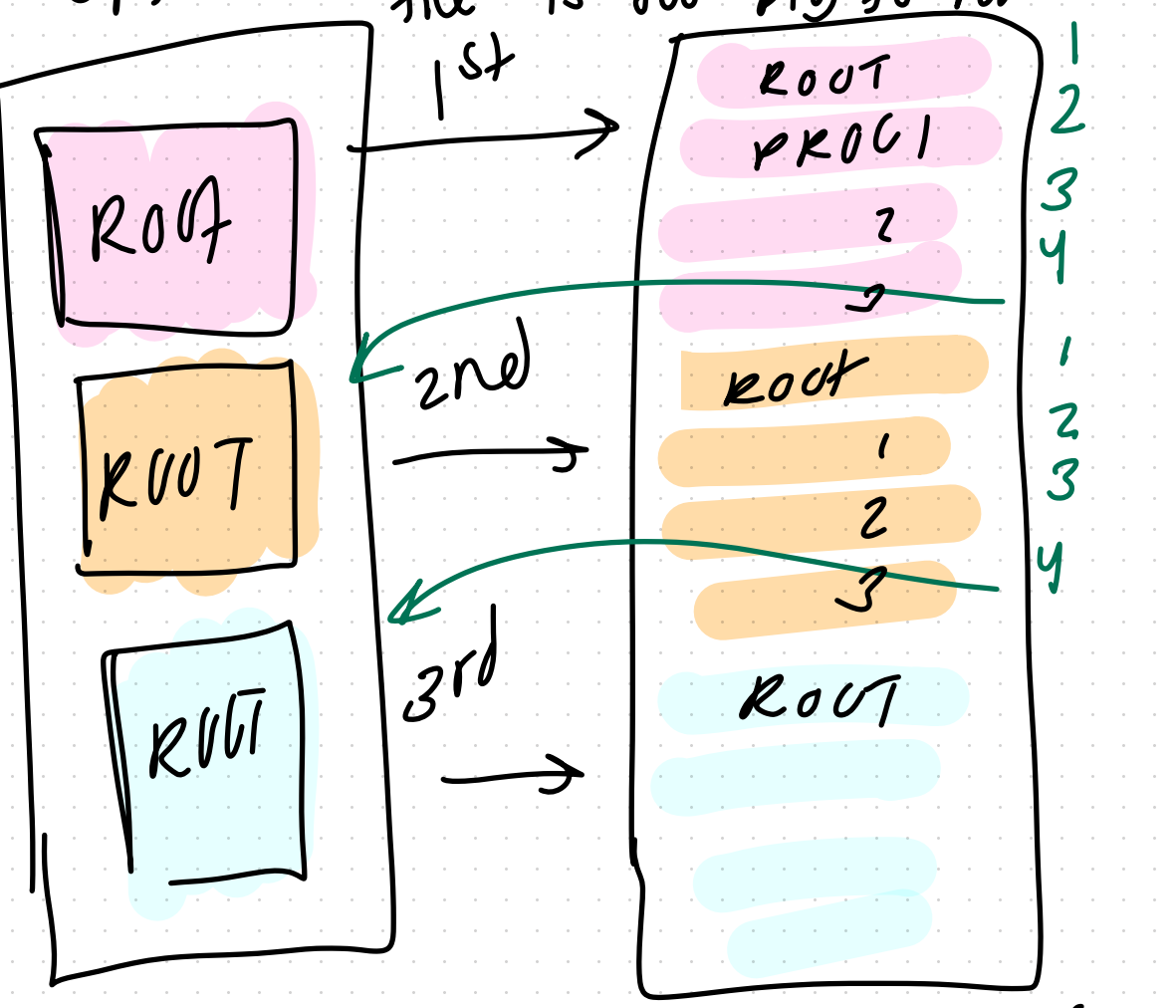
" think about scattering & send recv, we took window split and put into contiguous obj.

hexdump -e '20/4 "%4d"' -e '"\n"' window

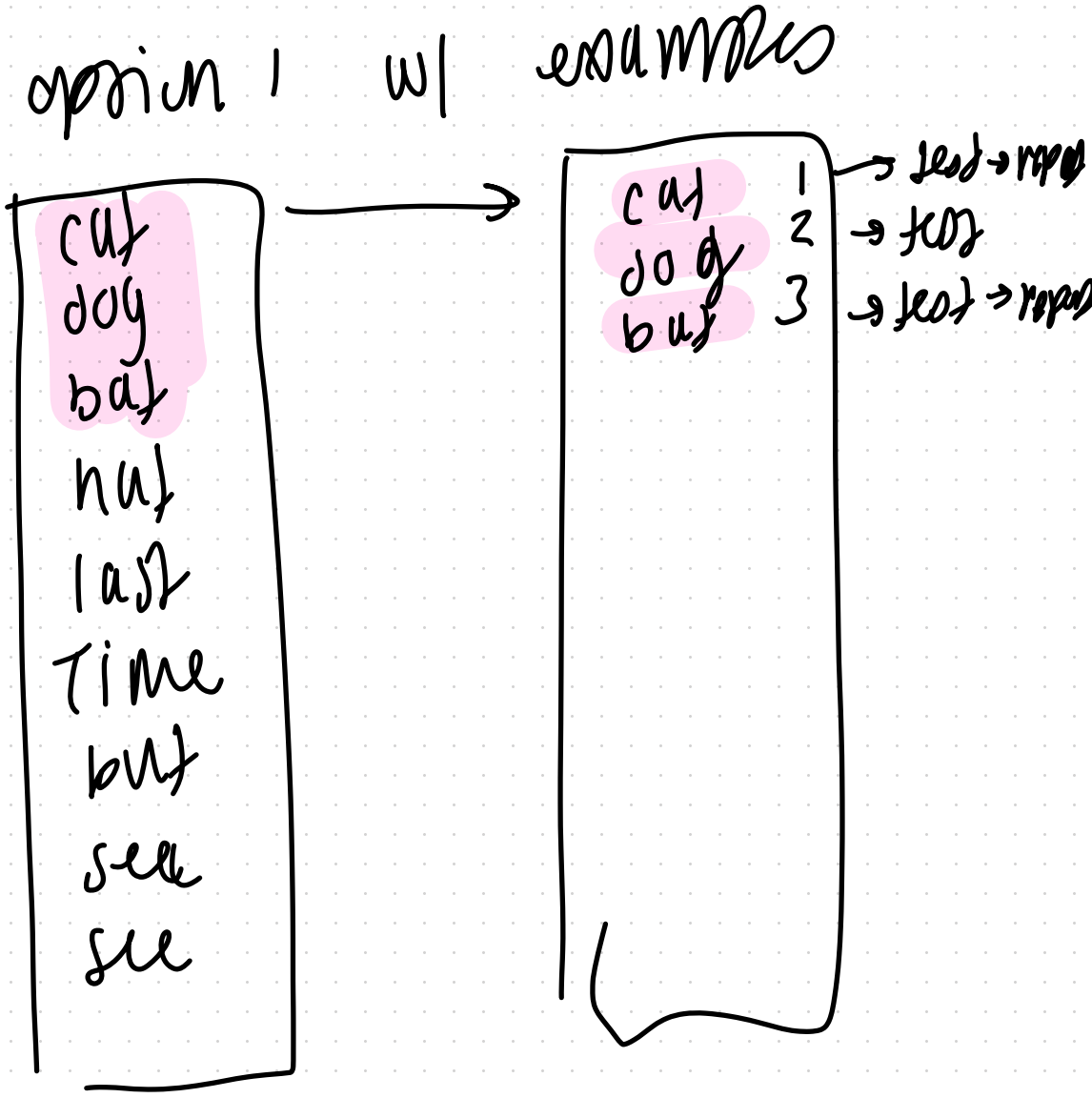
lscpu command

- take advantage of cache size and add the other for loop to optimize

option 1 - Might have to do if file is too big to read at 1



ROOT reads x amount of times 1 chunk into dict then scatters to every proc which runs to check if pswd found


option 1 w/ examples
cat
dog
bat
hat
last
time
but
see
see
cat 1
dog 2
but 3