We live in a competitive world...

This practical exposes system library tools for multi-process, multi-threaded, and elements of concurrent programming. Anything else is just fortunate. All starter files are posted on studres/.

Task 1: The Telephone Game (not Chinese Whispers)

Make a chain of **10** processes, by forking repeatedly. Each newly created process will:

1. Read a string sent from its parent process and write it to `stdout`.
2. Make a new child process.
3. Switch any two letters in the string.
4. Send the changed string on to its new child via a `pipe`.
5. (Optional Extension with Task 3.) Write to a file such that (i) all processes will have written to a single file, `rumours.out`, when complete and (ii) the changed strings appear in the reverse order from which they were generated.

The program should accept a series of words from `stdin`. For example,

```
> ./rumours I love fish and chips
```

The first child might then receive the string as, “`x love fish ani chips...`”. Note that string modifications must be made by the parent *after* the child has been forked.

OUTPUT messages should adhere to the following format:

- (`stdout` only) – When a new process starts it should print its own pid, as well as its parent’s pid, e.g. `New process:  74; parent:  73`
- (Both `stdout` and `rumours.out`) – All other output should be prepended with the process’ pid, e.g. `pid 74:  [output]`

An example output is provided below. Any perceived formatting or ordering is just luck; no formatting across lines is required.

```
> ./rumours i love fish and chips
pid:27569 Swapped indices 14,0
New process: 27570, parent 27569
pid:27570 received string: d love fish ani chips
New process: 27571, parent 27570
pid:27570 Swapped indices 0,7
pid:27571 received string: f love dish ani chips
pid:27571 Swapped indices 7,14
New process: 27572, parent 27571
pid:27572 received string: f love iish and chips
pid:27572 Swapped indices 14,0
...
```

WARNING: Save frequently! When working with fork, your computer may behave badly and crash other applications.

Task 2: “Threadeo, threadeo, wherefore art thou?”

A common misconception is that ‘wherefore’ means *where*, when in fact it means *why*. In this task, in the context of multi-threading, both questions will be answered by tracing through code and generating a visual thread-join representation.

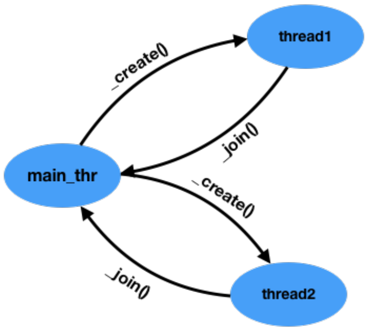
YOUR TASK: In your report, draw or generate the thread-join diagram for the following code:

```
void* worker(void *data){
  sleep(1);
  if(data) {
    pthread_join((pthread_t)data, NULL);
  }
  return NULL;
}

int main(){
  pthread_t thread = (pthread_t)NULL;
  for(int i = 0; i < 5; ++i){
    pthread_create(&thread, NULL, worker, (void*)thread);
  }
  pthread_join(thread, NULL);
}
```

Consider an example modelled from `threads1.c`, slide 22, lecture 2019-04-03-SP06:

```
void *thread(void *arg){
  // do something
  return NULL;
}

int main(int argc, char* argv[]){
  pthread_t tid1,tid2;
  pthread_create( &tid1, NULL, thread, NULL );
  pthread_create( &tid2, NULL, thread, NULL );
  pthread_join( tid1, NULL );
  pthread_join( tid2, NULL );
}
```

The corresponding thread-join diagram looks as follows:



Include a justification. Why is your diagram correct? What is the evidence? (A diagram is insufficient; with an explanation is better; only with evidence can the answer be fully correct.)

Task 3 [Extension]: A real "Shadow Breaker"

NOTE: Start work anywhere, but `crypt_r()` is available only on lab (Linux) machines.

Passwords should *never* be stored on a system in plain text. Instead, we store hashed versions of passwords. For example, in modern Linux systems, user information is stored in `/etc/passwd`, while corresponding passwords are hashed into `/etc/shadow`[1]. When a user logs in, the password given is hashed and compared against the stored hash.

Given a good hash function it is (believed to be) hard or impossible to retrieve the original input from the hashed value efficiently. However, it is possible to expend CPU time to try every possible password until one hashes to the target hash[2].

`crypt_r()` is the re-entrant, i.e. thread-safe, version of `crypt()` from the `crypt.h` library. The function takes three arguments: (i) the string to hash; (ii) a 'salt' string that is said to 'flavour' the input, **always use "`cs2002`" for this practical**; (iii) a `struct crypt_data`, in which the `initialized` member must be zero'ed (although best to zero the whole struct). Full documentation is available in the man page, `$ man crypt_r`. Here is an example use:

```
// Make sure to include <crypt.h>
// and link with "-lcrypt"
struct crypt_data cdata;
cdata.initialized = 0;

const char *hashed = crypt_r("my1stString", "cs2002", &cdata);
printf("hash of 'my1stString' = %s\n", hashed);

hashed = crypt_r("my2ndString", "cs2002", &cdata);
printf("hash of 'my2ndString' = %s\n", hashed);
```

The code above produces the following:

```
hash of 'my1stString' = cs6eF1v.Cgmq.
hash of 'my2ndString' = cskyYQmzegzV6
```

YOUR TASK: Given a hash, recover the original password. Alongside the hash, a partial password and password length will be provided. Input to the program comes from `stdin` with space-delineated entries, as follows:

- Username - 1 to 8 characters
- Password hash - 13 characters
- The password prefix (i.e. the part that is known), with periods representing the unknown characters. Password lengths (len) are always 1 to 8 characters in length, consisting of x characters and y periods, where $x + y = len$.

So, for example, if the provided `create_examples` tool is used to generate a password, then

```
> ./create_examples 1 15 23987 > pwd_hashes.txt
> cat pwd_hashes.txt
jasmin xxDxBTgGws2Bo qhz.....
> ./shadow_breaker < pwd_hashes.txt
```

A suite of helper functions are provided in `output_helpers.h` for output, and `utils.h` for password processing, each with corresponding `.c` files for your perusal. **Please use them.**

[1] 'Shadow' password files can still be stolen or breached. Breaches of users' online accounts can be checked using https://monitor.firefox.com

[2] In practice, the energy cost alone makes this impractical for most purposes, never mind the time and resources!

How to Proceed

First, it is always a good idea to write a single-threaded solution before attempts at multi-threading. Interestingly, this is faster than any parallel solution on a single-core cpu!

Now use multiple threads to speed up recovery. Edit `start()` so that the main thread can:

1. read the line of input from stdin;
2. print "`Start <username>`" (use the helper)
3. spawn `thread_count` threads (command line argument, otherwise default is 4) and
   - pass to each thread its own thread index starting from 1,
   - provide the hash and partial password including periods;
4. wait for the password to be recovered, then output a summary (see 'helpers' below).

Worker Threads, once spawned, must distribute the search space between them into `thread_count` ranges of equal size. For example, 3 unknown characters means $26^3 = 17576$ possible passwords. Four worker threads would split this range as follows,

- Thread 1: 0...4393 (or aaa...gmz)
- Thread 2: 4394...8787 (or gna...mzz)
- Thread 3: 8788...13181 (or naa...tmz)
- Thread 4: 13182...17575 (or tna...zzz)

The order of operations for each thread (roughly) is described by the following:

1. Read or receive the hash and partial password.
2. Identify its equal-sized portion of the possible password range to evaluate by,
   - using `getSubRange()` with its thread index to identify its range;
   - then using `setStringPosition()` to set the start of the range.
3. Output a thread_start message to the screen (use the helper).
4. Look for the password within the range.

When a worker thread finds the correct password, it must indicate to other threads to stop working on the task. This can be implemented with a simple flag variable that each thread checks on each iteration. Since all threads read this variable and any can write to it, access must be properly synchronized.

When any worker completes the task, each worker must print a summary message using `print_thread_parr_result`

When the task is complete, the main thread should output a summary message. For this purpose `print_parr_summary()` is provided; proper count and time keeping variables may be supplied with dummy values if you wish.

Promises, Hints and Guiding Observations

- $2 \leq$ `thread_count` ≤ 13.
- **All code should be restricted to `shadow_parr.c/h`** (as well as other files of your own creation, if you wish).
- **Globals are permitted** to exchange information between the main thread and workers; although bad practice, it may reduce the complexity (but be aware that race conditions may need to be handled).
- All other functions, structs, etc., are left to your own devising.

Questions and Challenges to Ponder Upon Completion

There are many possible extensions to Task 3. None are recommended as valuable use of your time given the effort they require, but they are interesting nonetheless.

- Multiple passwords. For this purpose a *synchronized* queue implementation is provided.
- If there are multiple passwords, does it make more sense to parallelize the task, or allocate each password uniquely to a thread?
- Could you create a synchronized queue of your own? (Source code left visible for your perusal.)

Ranges of Assessment

Assessment of tasks is cumulative, as described below. In each tier, a well-written report is assumed.

- **Up to 13.** A good implementation of Task 1.
- **Up to 17.** A good implementation of Task 1 in addition to a rigorous presentation that answers Task 2.
- **Up to 20.** Task 3, with Tasks 1 and 2 as prerequisites for consideration.

In the absence of Task 3, note that

- step 5 marked as optional in Task 1 is mandatory;
- Tasks 1 and 2 will be assessed against the highest standards of well-written code, appropriate comments, also full checking of errors, bounds, and return values. (This is much less work than required for completion of Task 3.)

Assessment & Rubric

Generally speaking, source and reports will be graded primarily along the following criteria:

Scope	The extent to which work implements the features required/specified
Correctness	The extent to which work is consistent with the specs and bug-free.
Design	The extent to which work is well written, ie. clearly, succinctly, elegantly, or logically.
Style	The extent to which work is readable, eg. comments, indentation, apt naming

There is no fixed weighting between the different parts of the practical. So that students might be better prepared for subsequent practicals, general feedback will be offered according to guidelines published in the Student Handbook at,

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html

Work must be submitted via MMS by the deadline. The standard lateness penalties apply to coursework submitted late, as indicated at

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

As a reminder, the relevant guidelines on good academic practice are outlined at: http://www.st-andrews.ac.uk/students/rules/academicpractice/