pfind

Tester

Python 3 tester that randomly generates filesystems for the following cases: * Normal case (without links or unsearchable directories) * With links (30 percent of created files will be links, 70 percent will be regular files) * With unsearchable directories (directories where I remove `read` permission) * All of the above (both with links and with unsearchable directories)

For every such case, the tester checks that the correct amount of files and link names contain our search term and that the correct amount of "Permission Denied: \ <DIR>" were printed during our run.

In addition, it ensures that no line was printed out more than once (could mean that multiple threads handled the same directory).

If tester says that you printed to many rows (but no duplicates) it might mean you don't check permissions well or that you also print directory names that match the search term, and not only files.

To recognize a deadlock, every run is allowed a timeout and if that is reached, the tester raises an error. After the error is recognized, the file system will not be deleted so you can run it yourself and debug your program. To help do that, the tester will print the command that found the bug. Because of the processor's scheduling mechanism, it might take a few tries until you hit the same racing condition as the tester reached.

Instead of manually rerunning it, you can do the following, assuming the command is CMD: * For MacOS: `repeat N {CMD}` . For example `repeat 5000 {./pfind /path/to/dir "pdf" 2}` * For bash: `for {1..N}; do CMD; done;` . For example `for i in {1..5000}; do ./pfind /path/to/dir "pdf" 2; done;` * In nova, you must use bash so before using this command, run `bash` and then run the loop

Now that you've finished reading everything, just run: `python3 tester.py`

IMPORTANT AND VERY USEFUL REMARK:

A new feature has been added to the tester: If a timeout is reached when running the job, that either means that you have a deadlock or that you have an infinite loop. So now if it fails, it will print the whole output. But because the tester passes only outputs that contain ONLY allowed format (either a path or Permission denied), than we couldn't add any debug prints. Now you can! To any print you add for debugging purposes, run the tester with `--ignore-debug-prints` and add `***` somewhere inside the `pfind` 's prints and the tester will ignore this line when it checks for correct results, but print it out when it prints the output after a failure.

So run `python3 tester.py --ignore-debug-prints` If it has too many logs, redirect the output to a file and then analyze the file's content. You can redirect as following: `python3 tester.py --ignore-debug-prints > debug.log`

If you passed everything, pass also the argument `--hard` and that will retry many times, and thus increase the chances of finding a bug, because finding a bug could be a matter of chance (processor's scheduler has it's own rules that we cannot control)

Personal steps

Step I

First implement pfind without parallelism. Once it works, check for memort leaks by first compiling with `-g` flag and then running valgrind:

```
gcc -Wall -std=c11 -g pfind.c -o pfind
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --verbose ./pfind /Users/sid/Documents/studies "pdf" 1
```

And hope it prints out the following: `ERROR SUMMARY: 0 errors from 0 contexts` .

If it doesn't, start debugging and fixing.

Step II

Start implementing parallelism. You'll probably have deadlocks at first so you'll want to debug it. For simplicity, try first debugging it with parallelism 2.

Add prints EVERYWHERE (especially before / after EVERY lock). To be able to differentiate between different threads easily, you'll need to print the thread ID. To do so, add the following func (this assumes a count variable `runningThreads`):

```
#include <stdarg.h>
#include <stdint.h>

long getNanoTs(void) {
    struct timespec spec;
    clock_gettime(CLOCK_REALTIME, &spec);
    return (int64_t) (spec.tv_sec) * (int64_t) 1000000000 + (int64_t) (spec.tv_nsec);
}

char *debugFormat = "[%02x] : %lu : %d : ";
char *debugLevel = "***";

void debugPrintf(char *fmt, ...) {
    va_list args;
    va_start(args, fmt);
    char *placeholder = malloc(strlen(debugFormat) + strlen(debugLevel) + 1);
    char *newFmt = malloc(strlen(debugLevel) + strlen(fmt) + 1);
    snprintf(placeholder, strlen(debugFormat) + strlen(debugLevel) + 1, "%s%s", debugLevel, debugFormat);
    snprintf(newFmt, strlen(debugLevel) + strlen(fmt) + 1, "%s%s", debugLevel, fmt);
    pthread_mutex_lock(&printLock);
    printf(placeholder, pthread_self(), getNanoTs(), runningThreads);
    vprintf(newFmt, args);
    pthread_mutex_unlock(&printLock);
    fflush(stdout);
    free(newFmt);
    free(placeholder);
    va_end(args);
}
```

And use this INSTEAD of `printf`. You can use this function also with parameters (just like `printf`, for example `debugPrintf("Handling directory: %s", directory)`). This will format the prints in the following format:

```
***[b9f4000] : 1609272424809982000 : 6 : ***unlocked queue for enqueueing (read/write).
***[c1bb000] : 1609272424810261000 : 6 : ***(isDone) unlocked queue read lock queueRWLock. will return 0
***[b9f4000] : 1609272424810638000 : 5 : ***(isDone) locking queue read lock queueRWLock
***[9acc000] : 1609272424811054000 : 6 : ***(isDone) unlocked queue read lock queueRWLock. will return 0
***[af0b000] : 1609272424811419000 : 6 : ***(isDone) locking queue read lock queueRWLock
***[9acc000] : 1609272424811728000 : 7 : ***locking for cond var dequeueing
```

[threadId] : timestamp : message

That way you can easily differentiate between what happens in which thread.

I've seen cases that the timestamps could be in the wrong order, so if the order is important to you (and it should be in many cases), make sure you analyze the logs in the correct order

Remarks

- Do not try composing the queue lock of multiple sub locks (for example one for the items and one for the queue size). We use `pthread_cond_wait` and this receives only one mutex, so this must be the only mutex locking the queue.
- If except for `enqueue` and `dequeue` you also have a `getQueueSize` function, then you could use Read/Write locks. `queue` and `enqueue` will WriteLock but `getQueueSize` will only ReadLock and this will make `pfind` more scalable because reading the queue size won't block other threads that only just want to read the queue size.
    - Note that `pthread_cond_wait` receives a mutex and not a RWLock so you must create an additional mutex for this, for example `pthread_mutex_lock(&queueLock); while (unsafeGetQueueSize() == 0 && runningThreads > 0) { pthread_cond_wait(&queueConsumableCond, &queueLock); } pthread_mutex_unlock(&queueLock); pthread_rwlock_wrlock(&queueRWLock); char *path = unsafeDeQueue(); pthread_rwlock_unlock(&queueRWLock);`

ReadWrite Locks

They works very similarly to mutexes. Use the following functions:

```
 pthread_rwlock_t queueRWLock;
pthread_rwlock_init(&queueRWLock, NULL);

pthread_rwlock_rdlock(&queueRWLock);
// ... read only actions ...
pthread_rwlock_unlock(&queueRWLock);

pthread_rwlock_wrlock(&queueRWLock);
// ... read/write actions ...
pthread_rwlock_unlock(&queueRWLock);

pthread_rwlock_destroy(&queueRWLock);
```