searchmount

Search Directories for UNIX

Ashley Smith (ashley3@MIT.edu)

Jackson, 2PM

6.033 DP1: Report

3/21/2014

1 Introduction

This report offers a solution to the current inefficiency of repeated searches in UNIX. Currently, users perform searches with the find command but the results of these searches cannot be saved for future use. Windows and iOS solve this problem with search directories. Search directories are special virtual directories that contain the results of a file search query as symbolic links to the original files. This report details an implementation for search directories in UNIX.

1.1 Design Overview

Search directories will take the form of a UNIX command-line tool called `searchmount`. `searchmount` takes in an expression similar to that which the find command uses and creates a new virtual directory for the results of the search. These results are stored in a series of on disk databases that contain information about the search along with information about each entry in the resulting directory. The recently used search directories are also cached in memory for quick access.

`searchmount` uses *inotify* to handle changes made to files in the searched directories. `searchmount` creates *inotify* watches for the directories in the search path which listen for changes. `searchmount` also creates a background thread which checks for any changes and updates the corresponding databases to keep search directories up to date.

1.2 Trade-Offs and Design Decisions

A major design decision made in `searchmount` is to store the search directories on disk. Users would expect their recently mounted search directories to still exist when they restart their computers. While storing search directories on disk is not extremely efficient in terms of access time, it is less costly than recreating search directories on startup. Recreating search directories on startup could take a lot of time as each search would need to be repeated. To accelerate access times associated with storing search directories on disk, recently used search directories are cached.

An additional major design decision made in `searchmount` is to immediately update search directories when a change is made to a file. Users want their files to be up to date. While it would be less costly in terms of processing time to update search directories on an hourly basis or even a daily basis, we chose to optimize based on user experience. This additional processing time is well worth the result of having constantly up to date search directories for users.

2 Design

The main components of `searchmount` are on disk databases which store search directory results and relevant metadata, *inotify* to watch target directories for changes, and the provided application programming interface (API) for access to previously created search directories.

2.1 On Disk Databases

On disk databases are used to store the results of a search. Search directories are stored in three different types of databases: DirPointer databases, DirEntryPointer databases, and a Search

Directories Database. The following sections describe these databases in more detail and explain how they are created and removed.

2.1.1 Storing search directories

Search directories are stored in persistent databases on disk. In order to store the results of a search, we define two data structures stored in databases: DirPointer and DirEntryPointer. When a DirPointer database or DirEntryPointer database is read, it is cached in memory to ensure fast subsequent access to that search directory.

There are three types of databases used by `searchmount`: Search Directories Database, DirPointer databases, and DirEntryPointer databases. Figure 1: On Disk Databases shows these data structures and the data they hold.

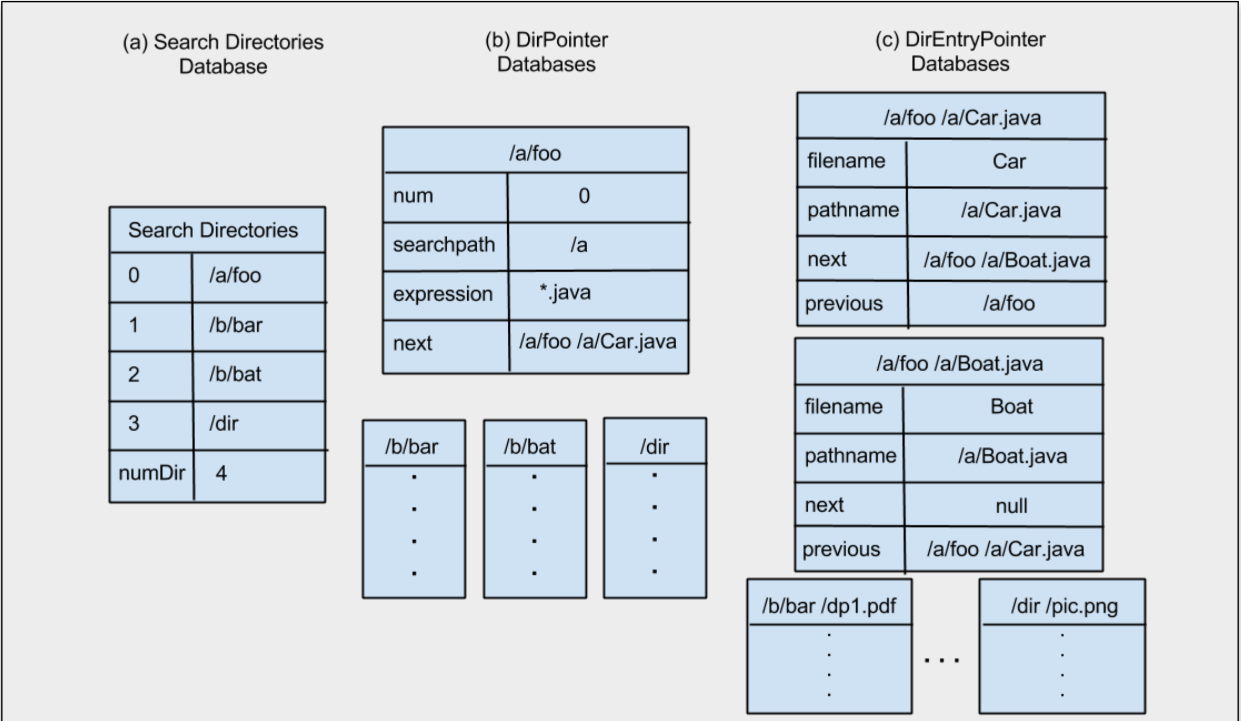


Figure 1: On Disk Databases
(a) The search directories database provides the ability to iterate through the search directories. (b) DirPointer databases contain important metadata about each search directory along with a next field with a value of the first entry in the search directory. (c) DirEntryPointer databases act as a doubly linked list with each DirEntryPointer database having the name of the next DirEntryPointer database in the search directory or null if it is the last entry in that search directory.

`searchmount` has exactly one Search Directories Database. As new search directories are created, the search directories are added to this database. This database will be used to create *inotify* instances on startup as described in section 2.2.2 Creating new *inotify* instances and watches.

`searchmount` creates one DirPointer database for each search directory. DirPointer databases contain the information used to create the search and the name of the first DirEntryPointer database to represent the first entry in the search results.

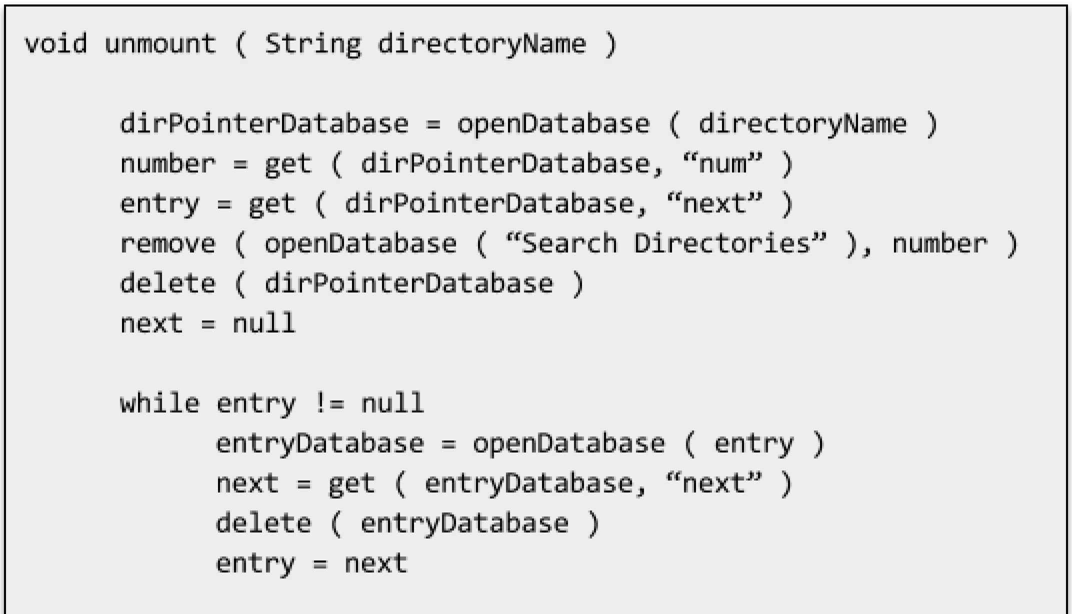
Lastly, each DirEntryPointer database represents one entry in the search results. DirEntryPointer databases contain metadata about each entry along with the name of the previous and next DirEntryPointer databases in the search results.

2.1.2 Creating mountpoints

When a mountpoint is created, `searchmount` creates a background thread to create the directory and its corresponding DirEntryPointer databases. This thread calls the find -P command to get all files that should be virtually linked to (note: the searched files do not include virtual directories). The thread then sets up the database to contain these results by initializing a DirPointer database and many DirEntryPointer databases, linking them together, and putting a link to the DirPointer database in the Search Directories database. Finally, this background thread is responsible for setting up *inotify* watches for all of the directories in the search path. This process is detailed in 2.2.2 Creating new *inotify* instances and watches.

2.1.2 Removing mountpoints

When a mountpoint is unmounted, `searchmount` creates a background thread to remove the directory and its corresponding DirEntryPointer databases. The background thread is given the name of the search directory to be removed and the following method is called:



```
void unmount ( String directoryName )

      dirPointerDatabase = openDatabase ( directoryName )
      number = get ( dirPointerDatabase, "num" )
      entry = get ( dirPointerDatabase, "next" )
      remove ( openDatabase ( "Search Directories" ), number )
      delete ( dirPointerDatabase )
      next = null

      while entry != null
            entryDatabase = openDatabase ( entry )
            next = get ( entryDatabase, "next" )
            delete ( entryDatabase )
            entry = next
```

Figure 2: Pseudocode for Unmounting
This pseudocode removes the specified search directory from the search directories database, deletes the DirPointer database, and recursively deletes all the DirEntryPointer databases.

2.2 *inotify*

`searchmount` utilizes *inotify* to listen for changes to files in each directory in each search path. When an *inotify* watch notices a change, it locates the added/updated file and writes the change to a file.

2.2.1 Watching for changes in search directories

inotify is a file change notification system. *inotify* allows applications to watch directories for a specified set of changes. In `searchmount`'s case, *inotify* watches each directory in each search path of each search directory for changes in metadata, deletions of files, and moves of files in or out of the directory. When any of these changes occur, *inotify* writes the event to a file to be responded to by `searchmount` [1]. This process is shown in Figure 3: *inotify*.

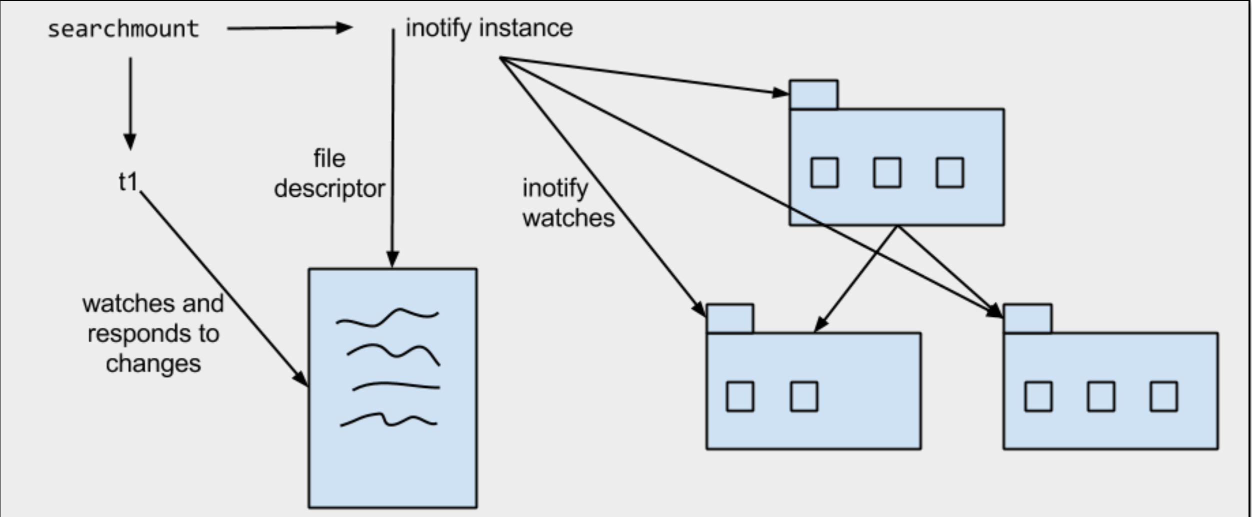


Figure 3: inotify
`searchmount` creates an *inotify* instance for each search directory. Each *inotify* instance has multiple *inotify* watches, one for each folder in the search path. When *inotify* notices a change in any of the files being watched, *inotify* writes the change to a file. `searchmount` starts a thread when it creates the *inotify* instance. This thread watches the file and updates the relevant databases after each change.

2.2.2 Creating new *inotify* instances and watches

inotify instances and *inotify* watches will be created on startup and when new search directories are created. On startup, *inotify* instances and watches need to be created for each search directory that currently exists. searchmount iterates through the Search Directories database, by getting zero to numDir, and creates *inotify* instances and watches for each as demonstrated in Figure 4: Using *inotify*. This process for creating *inotify* instances and watches is the same for when new search directories are created.

```
void createInotifyInstance ( String directoryName )
    fileDescriptor = instantiate inotify instance
    instantiate in memory hashmap for watch descriptors and directories
    for each directory and sub-directory (excluding virtual directories) in the search path:
        watchDescriptor = add_watch(fileDescriptor, directory, IN_ATTRIB | IN_MOVED_TO |
                    IN_MOVED_FROM | IN_DELETE )
        add watchDescriptor, directory to the hashmap
```

Figure 4: Using inotify
This pseudocode demonstrates the process for creating new *inotify* instances and watches. The process involves adding a new watch for each directory in the search path and saving a reference to the watch descriptor in an in memory hash map. After this code block, the current thread will begin polling the file descriptor for new events and handling them as described in 2.2.3 Updating search directories.

2.2.3 Updating search directories

The background thread created by `searchmount` to create the *inotify* instances and watches also takes responsibility for constantly monitoring the file for any events and updating the databases accordingly.

`searchmount` can determine if a DirEntryPointer database exists and is in the linked list by opening the directory with the name <search directory mountpoint> + " " + <file name> and checking if a get for the pathname key returns a value. The search directory name is given to the thread when it creates the *inotify* instance and the file name is provided in the *inotify* event.

2.2.3.1 Changes in metadata

If the changed file's metadata matches the expression and is not already in the DirEntryPointer database linked list, it will be added to the start of the linked list. If it does not match and previously did, the DirEntryPointer database will be removed from the directory entry linked list and deleted.

2.2.3.2 Deletions of files

If the event signals the deletion of a file, `searchmount` deletes the associated DirEntryPointer database, if it exists, and adjusts the next and previous values for the surrounding DirEntryPointer databases in the linked list.

2.2.3.3 Moves of files in or out of the directory

If a file is moved into the directory and matches the expression for the search directory, a DirEntryPointer database is initialized and added to the start of the search directory's linked list.

If a file is moved out of the directory and matches the expression for the search directory, the associated DirEntryPointer database is removed from the linked list and deleted.

If the file moved into the directory is a folder, a new watch is created for that folder (see 2.2.2 Creating new *inotify* instances and watches for more details).

Note: There is a possibility of events being lost due to an *inotify* queue overflow. Because `searchmount` immediately handles events, this probability is very low and not addressed in this implementation.

2.3 Application Programming Interface

Users create new directories using the `searchmount` command and access previously created directories through an API.

2.3.1 Creating new and removing old directories using `searchmount`

`searchmount` offers users the following two commands to create and remove search directories:

```
searchmount mountpoint searchpath [expression]
```

```
searchmount –u mountpoint
```

`searchmount` creates a new directory `mountpoint` and makes the new directory appear to contain symbolic links to files under `searchpath` matching the search query given in `expression`. The second command removes the `mountpoint` directory.

Internally, when `searchmount` is run, the following method will be called to create the new directory:

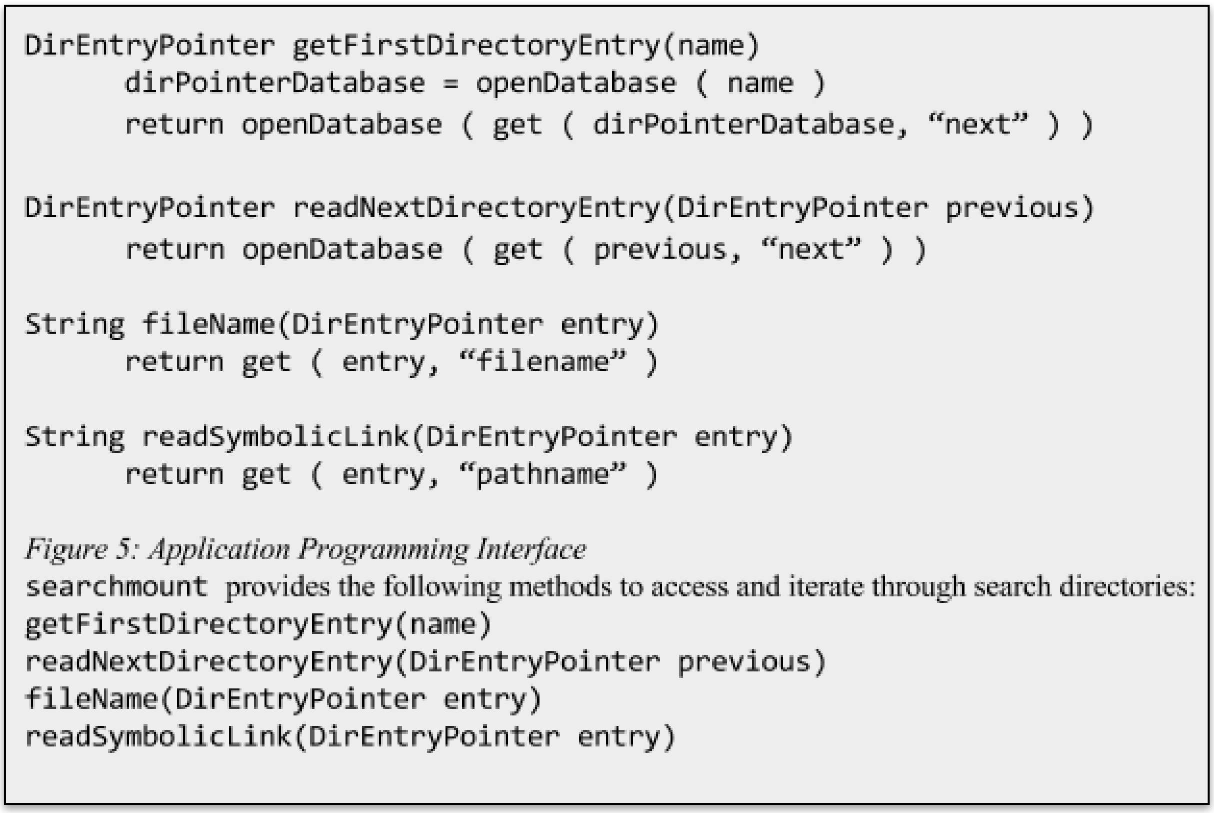
```
createDirectory(name, searchPath, expression)
```

`createDirectory` starts by checking if the `mountpoint` already exists. If the database lookup with key `name` returns `null`, searchmount creates a background thread to create the directory and its corresponding DirEntryPointer databases, as described in section 2.1.2 Creating mountpoints.

When the second command is run, `searchmount` creates a background thread to remove the directory and its corresponding DirEntryPointer databases, as described in section 2.1.3 Removing mountpoints.

2.3.2 Accessing previously created directories through the API

`searchmount` provides the API in Figure 3: Application Programming Interface to access previously created directories:



```
DirEntryPointer getFirstDirectoryEntry(name)
      dirPointerDatabase = openDatabase ( name )
      return openDatabase ( get ( dirPointerDatabase, "next" ) )

DirEntryPointer readNextDirectoryEntry(DirEntryPointer previous)
      return openDatabase ( get ( previous, "next" ) )

String fileName(DirEntryPointer entry)
      return get ( entry, "filename" )

String readSymbolicLink(DirEntryPointer entry)
      return get ( entry, "pathname" )
```

Figure 5: Application Programming Interface

`searchmount` provides the following methods to access and iterate through search directories:

`getFirstDirectoryEntry(name)`
`readNextDirectoryEntry(DirEntryPointer previous)`
`fileName(DirEntryPointer entry)`
`readSymbolicLink(DirEntryPointer entry)`

3 Analysis

The following analysis demonstrates that searchmount runs in linear time for the three common use cases we analyzed. The major efficiency problem discovered is in creating *inotify* watches. Additionally, `searchmount` has a few scalability problems. These problems are addressed in Section 3.2 Overall Analysis.

3.1 Use Cases

The following sections analyze three common use cases to determine how well `searchmount` performs.

3.1.1 Searching small target directories
`searchmount` runs in linear time in this use case. The majority of the time will be spent doing the `find` and the `put` for all of the key value pairs. Performing the find will take linear time with respect to the number of files in the target directory, and performing the `put` will take linear time with respect to the number of files in the search results. As the target directory has a modest number of files, linear time will still be quite fast.

3.1.2 Searching the system's root directory
In this use case, linear time is not very fast due to the size of a system's root directory. As stated in 3.1.1, the majority of the time will be spent doing the `find` and the `put` for all of the key value pairs. Fortunately, both time-consuming processes will be run in a background thread so that the system remains available for use during this time.

Unfortunately, not all search results will be available to the user when the terminal returns. If the folder is accessed before all of the results are available, the terminal will display the results that have been found so far (if any) and a line of text informing the user that the folder is in the process of being updated.

3.1.3 Mounting search directories automatically at startup or login
Because search directories are saved on disk or created in a background thread, `searchmount` excels when mounting search directories automatically at startup or login. If `searchmount` is asked to mount a search directory that is already mounted, no additional functionality needs to be performed. If the search directories being mounted at startup or login have been unmounted or have not yet been created, `searchmount` will run in linear time and in the background as explained in sections 3.1.1 Searching small target directories and 3.1.2 Searching the system's root directory.

3.2 Overall Analysis
While `searchmount` is efficient for accessing search results, creating *inotify* watches is inefficient. Additionally, `searchmount` does not scale well due to space constraints when a user creates many search directories.

3.2.1 Efficiency
From a user's perspective, `searchmount` is efficient when accessing search results. Accessing DirPointer databases and DirEntryPointer database using the API takes constant time as all of the information is stored in the databases. Storing search results takes constant space with respect to the number of files in the search directories and the number of search directories.

Creating *inotify* watches, however, is fairly inefficient. searchmount needs to traverse all of the directories and subdirectories in the search path again which takes linear time with respect to the number of files in the directory. Additionally, there is an *inotify* watch for each directory for each search directory whose search path contains those directories, meaning some directories will likely have more than one *inotify* watch listening for changes. These duplicate *inotify* watches could be reduced to improve efficiency in terms of speed and space.

3.2.2 Scalability
Unfortunately, this design is not very scalable in its current state. Each search directory corresponds to one thread. Thus after a user creates a few search directories, these threads will be

taking up a lot of processing power. Search results are also stored on disk which is limited space. Once a user runs out of space to store these results, the user will be unable to mount new search directories until he or she unmounts another. These scalability problems will need to be resolved before this design for search directories in UNIX can be implemented.

4 Conclusion

This document details an implementation for efficient search directories in UNIX that contain the results of a search and offer access to these results in the future without having to repeat the search. `searchmount` provides two UNIX commands for both advanced and novice users to create and remove these directories. `searchmount` stores these directories in on disk databases and utilizes *inotify* to keep these search directories up to date.

There are two main problems that must be resolved before this design can be implemented. First, the scalability issues discussed in section 3.2.2 Scalability need addressed. Specifically, the case where the user runs out of disk space and wants to mount a new search directory needs addressed, and the number of threads needs to be reduced. Second, the number of *inotify* watches needs to be reduced. This reduction can be done by combining the *inotify* watches of common search paths. After these problems have been resolved, `searchmount` can be implemented.

5 Acknowledgements

I would like to thank Katrina for her invaluable instruction on UNIX. I would also like to thank Pratiksha for her feedback on the DP1 proposal. Finally, I would like to thank Nora Jackson and Jessie Stickgold-Sarah for their communication instruction and for their feedback on the DP1 proposal and draft of this report.

6 References

R. Love, “Kernel Korner – Intro to inotify,” in Linux Journal, no. 139, 2005.

D. M. Ritchie, K. Thompson, “The UNIX Time-Sharing System,” in Communications of the ACM, vol. 17, no. 7, 1974.

J.H. Saltzer and M.F. Kaashoek, M. Frans, Principles of Computer System Design. Waltham, Massachusetts: Morgan Kaufmann, 2009.

WORD COUNT: 2500