Alex RInehart
Ben Myshkowec
Group E3

## Summary

For our final project, we created a kernel module that creates and solve mazes based on parameters provided by the user. We chose this project because the algorithms for creating mazes are well-established and Alex was familiar with some of them. Transferring this functionality to the kernel space provided a new challenge to perform a familiar task without the convenience of standard libraries. In addition, this kernel module allowed to to practice concepts learned in this class such as kernel modules, basic interprocess communication, and general bash interaction.

The workload was split so that Alex focussed on the creation of the maze, while Ben implemented the maze solving algorithm. Ben and Alex collaborated to create the input parsing code. Ben worked to simplify repeated tasks (creating bash scripts to run and install the kernel, for example). Ben also created the user space program that would interact with our kernel module and handle all of the inter-process communication. Due to this split of the workload, the work ended up being split about evenly. While Alex's part may have been more difficult, he had familiarity with maze-generating algorithms, which balanced out.

## Challenges

We expected to run into problems with memory limits, as our code has relied on recursion from its inception. If the recursion failed on the solving side, we planned to find a solution by the use of a stack that Ben would have had to create. If it failed on the creation side, a new algorithm would have had to be researched. Fortunately, the only problems we actually ran into were bugs related to our code, and rarely pertained to the restrictions of implementing a kernel module.

One of the first challenges we ran into was memory. Specifically, after running into errors when performing recursion, we were suspected that the kernel module may not have had enough memory for the maze generating algorithm to run. After increasing the limit and discovering that our code still failed, we took a closer look and discovered inherent problems within it. Among these problems was an infinite loop with our recursion caused by incorrect comparison operators resulting in one condition being checked twice, and another never being checked. Since this operator was used to determine if our recursion had reached the base case, and could exit, we never did.

Upon fixing the infinite recursion problem, we discovered yet another program-crashing bug: a kernel panic. At this point in our development process, we attempted using a variety of debug tools, including GDB, to gain more information about our crashing kernel than "kernel panic". An hour of research told us that the easy way to accomplish this was to utilize a program that seemed to have been systematically removed form the internet (or at least was hosted at a dead link and never mirrored). The "complex way" to solve this problem involved a lot of concepts that we did not remotely understand.

In the end, we decided it would be a poor use of our time to tinker with kernel settings that we did not fully (or even vaguely) understand, and instead debugged with other methods. Specifically, Alex modified the code to run in user space and used GDB to find the source of the crash: a divide by zero error in our random function. This was temporarily patched by adding one to the denominator if it was zero, but was later remedied by using a more standardized random algorithm (the kernel's get_random_bytes() method) instead of the poor version we cobbled together.

Meanwhile, Ben's problems were much simpler, and typically presented themselves in minor errors in brace matching. These were easily remedied with sleep and GCC. The subtlest bug Ben discovered was that the maze seemed to look mirrored along the diagonal. This turned out to be caused a switching of the parameters "row" and "col" when indexing the maze's array, a bug that would rear its head later on.

After integrating Ben's code so that the project contained both maze creation and solving functionality, it was discovered that we could only create mazes that had the same height and width. If parameters were passed in such that the maze would be longer than it is tall (or vice-versa), it would populate so that it was square to one part, and then completely empty for the remainder of the maze. This bug also infiltrated Ben's maze solving code: when solving a maze of unequal dimensions, the solution would show that the exit path continued outside the bounds of the maze itself.

This was without a doubt the hardest bug to solve. GDB was not used this time, as the code compiled perfectly. Instead, print statements and major refactoring were used until it was discovered that there was a discrepancy in the naming conventions. Specifically, it was not consistent whether the maze was being invoked as [row][col] or [col][row]. while the [row][col] convention was common in the first part of the code, the [col][row] convention was prevalent throughout the code, and had to be adopted throughout the entire code. The biggest issue related to this bug was in functions named DivideVert() and DivideHorz(). The confusion resulted due to a confusing naming convention: specifically, it is unclear if DivideVert is dividing the maze by creating a vertical line, or dividing the maze vertically (by creating a horizontal line). As the code was worked on, this function was interpreted both ways, creating unbalanced or occasionally unsolvable mazes. A helpful comment and determined refactoring eliminated this. After much polishing, the code finally worked as intended. But our problems did not end there.

When moving on to test our code, we realized there was an issue with the echo command, manifesting itself as an invalid file format.We used echo to pipe input into our kernel module's device. Even when sending a properly formatted message, our program would crash. As it turns out, "echo" adds a newline to the end of input. Since this was unexpected, our code threw an error and discarded the input. looking at the man pages for echo revealed that this is a standard feature of echo (which makes sense for its typical use case). Luckily, the devs included a way to discard this behavior if it would be unwanted (such as now). the -n flag removes the newline and allows our code to compile and run without error.

A new can of worms was opened when we modified our code to interact with user space, instead of being solely in the kernel space. Most of these bugs were solved by reading man pages and finding the idiosyncrasies of functions such as fgets, fopen, and fputs. Solving these bugs reinforced both the concepts at hand, and the importance of man pages.

It has already been mentioned that our first bug was nothing more than a comparison operator resulting in an infinite loop. Unfortunately, this bug would show up again in the whitespace code. For a few hours, the effect of changing whitespace seemed to have the opposite effect than anticipated, and leaving it at a default of zero created an array that could be called a maze only by the most generous of onlookers. It was, of course, an easy fix, but it reinforced the possibility that Alex doesn't know his < from his >.

Our final bug was much more insidious. It presented itself after the decision was made to include the randomization of entrances and exits.Due to the aforementioned discrepancy between [row][col] and [col][row], there was one scenario where the randomly generated exit would not show up on the grid, leaving an unsolvable maze. The solution to this involved a close examination of each index in the array.

## How we did it

### Creation

The maze was stored in an a two dimensional character array, where a path was represented by the character 'a', and a walkway by the character ' '. Because this was a kernel module, and we did not have access to malloc(), we defined a maximum size of the array(50x50), and then chose to only populate the portion of the array that is specified by the user (determined by the parameters that they pass in). The parameters a user can specify are width, height, resolution, and whitespace, the latter of two are elaborated upon in the "Extra Features" section of this document.

The maze was generated using a modified version of a recursive division algorithm. Specifically, a recursive function called GenerateMaze() would take in the four corners of an array or subarray. It would subtract these to determine width and height, and then compare the sizes of the two. If both the height and width were within the smallest resolution desired, then the function would exit, having completed maze creation.

Otherwise, the function would split the array with a line. If the array was taller than it is wide, it would split it horizontally, where a split is performed by picking a random location and creating a wall there.. If it was wider than it was tall, a vertical split would be performed. If a square array was passed in, the split would be determined randomly. After the wall was created, a random location[1] within the wall would be selected, and a space would be formed there. This process would be repeated until the subarray was small enough to meet specifications

[1] In the case that a non-zero value for whitespace is used, there is a chance that multiple locations would be selected to be used for passages, but the base algorithm requires only one.

(determined by comparing to the chosen or default resolution). For a visual explanation of the maze generating algorithm, see the Appendix.

### Solving

The maze was was solved using a depth-first search to find a path throughout the array. The first step in this process is to examine the outside of the maze for a walkway. This would be designated as the entrance. From there, a recursive search would be performed, “travelling” in each direction where a walkway was found, until it found the outer limit of the maze. If a path was a dead-end, that particular branch of recursion would exit. Barring a dead-end, the square would be marked with a *, indicating that that is a cell on the correct path. Once this method returns the final time, the user is presented with a clear path throughout the maze.

There is one nuance with the maze-solving algorithm, and that is that it selects both elements of a double-wide path. This still results in a valid solution to the maze, however it is not guaranteed to be the shortest path through the maze.

### Parsing

Because a file passed into our program can contain either a solved maze or parameters for creating a maze, it is important to differentiate between these two cases. The way we do this is by reading the first character and checking if it is a number or a piece of the maze. If the first character is a walkway or a wall, the file will be parsed and stored into the maze array. After this, the solve_maze() function (described above) will be called.

Otherwise, the parameters (width, height, resolution, whitespace) will be stored into an array. The array is then parsed backwards, so that resolution, then height, then width are stored into temporary variables. These temporary variables can then be checked for validity and the maze creation function will be called.If an unexpected format is provided to the input buffer, the program will exit after displaying an error message. Usage information is provided when the kernel module is installed.

## Extra Features

While we initially set out to create a kernel module that would create a maze based on a user’s given height and width. A second minimum feature was the ability to solve any maze, whether it was created by our program, or just passed in.

In addition to meeting these goals, we were able to add two supplementary features to maze creation: resolution and whitespace. Instead of just size, users can specify four different parameters to generate a maze of their liking. Resolution determines how wide the walkways will be, and whitespace determines how frequently pathways show up. Resolution is an integer that should be within 3 and 50 (though a value larger than half the width or height will frequently result in an entirely open field). Whitespace is a percent, and should therefore remain within the

range [0,100]. Specifically, it is the chance that a given cell in a wall is instead made into an additional passage. Of course, resolution is a necessary parameter for generating a maze with this algorithm. We added the ability for the user to specify that resolution. A low resolution will create a classic winding maze, while a high resolution will create an open field. Similarly, a low whitespace will create a maze with only the minimum number of openings, while a high number will create output that more closely resembles an obstacle course than a maze. This allows for large flexibility that allows our kernel module to be applied in a variety of applications.

A third bonus feature that was added was the randomization of entrances. Initially, these entrances were hardcoded to the top-left and bottom-right corners of the maze. As the project progressed, we decided that didn't leave room for enough variety. Not wanting to deviate too far from the areas where a maze connoisseur would expect their entrances, we included code that would create an entrance and exit in one of two possible positions (four total possible openings).

Finally, we added a user-space program that handles all of the interprocess communication, and allows for the kernel module to be invoked. Of course, the kernel module can be interacted with directly, but the user space program provides arguments, including a help option. From the user space (but not from the kernel module directly) a user can specify an output text file that the maze or solution would be written to. If the user prefers, he or she can also specify that they would like their output in the form of a bitmap, and it will be stored in a bmp files format. The size of this bmp can be specified using a flag formatted like -b16.

## Results

As with all programs, it is difficult to test every input case that could crop up; it's easy for a pernicious line of code to slip through. Given the random nature of our project, this difficulty is amplified even further: an issue noted in one run may not reappear in subsequent tests, making duplication almost impossible.

Even with these obstacles, we feel confident that our code is robust enough to handle almost any input[2]. We have checked our validation procedures to eliminate corner cases, and have run our code numerous times with a variety of parameters, both valid and invalid. We have seen that our code reliably produces mazes that are both unique and solvable, or softly exits with an error message in the case of an invalid maze or bad file format. Our code is not entirely unforgiving. If provided with creation parameters that are close to being correct (but are perhaps incomplete), our code simply creates a maze using default values. No cases were found where our code crashes or provides invalid output.

## Discussion

[2] This is not a challenge, especially not to Kris.

Due to the graphical nature of our project, it is trivial to determine if it worked. If a maze is presented that contains both an entrance and an exit, and if that maze has a path from the start to that exit, our project worked. It is incredibly easy to verify a good result. In fact, part of our project does this for you. To verify our solution, it is simple necessary to follow the line of asterisks as they lead from entrance to exit which can be done at a glance.

The difficult part about verification of a solution arises when examining the output of a solved maze.Because we are not guaranteeing the shortest path, it is only necessary to verify that a path exists, and that we have selected one of those paths, This too can be done at a glance.

The tricky part of verification is ensuring that the solved maze we output is identical to the unsolved maze that was provided. Our code should (and does) return the same maze that was imported, but with the path highlighted. This is hard to verify at a glance (unless the differences are particularly egregious), but can be confirmed through the use of a diff.

## Conclusion

This project was a success. It did not take 12 hours a week, nor did it result in insurmountable confusion. The difficulty and time commitment were both appropriate. The project exemplified several pillars of operating systems that were covered in class such as terminal commands (cat, man, etc)., installing, removing, and creating a kernel module, basic interprocess communication (pipe, redirect, etc), and creating scripts to simplify repeated commands.. Through the course of this project, both Ben and Alex were able to gain familiarity working in a linux environment, creating, installing, and removing modules and devices, and troubleshooting in various environments. Their general coding abilities were also strengthened, as they grew less reliant on the standard libraries that had heretofore spoiled them.

Alex RInehart
Ben Myshkowec
Group E3

# Appendix

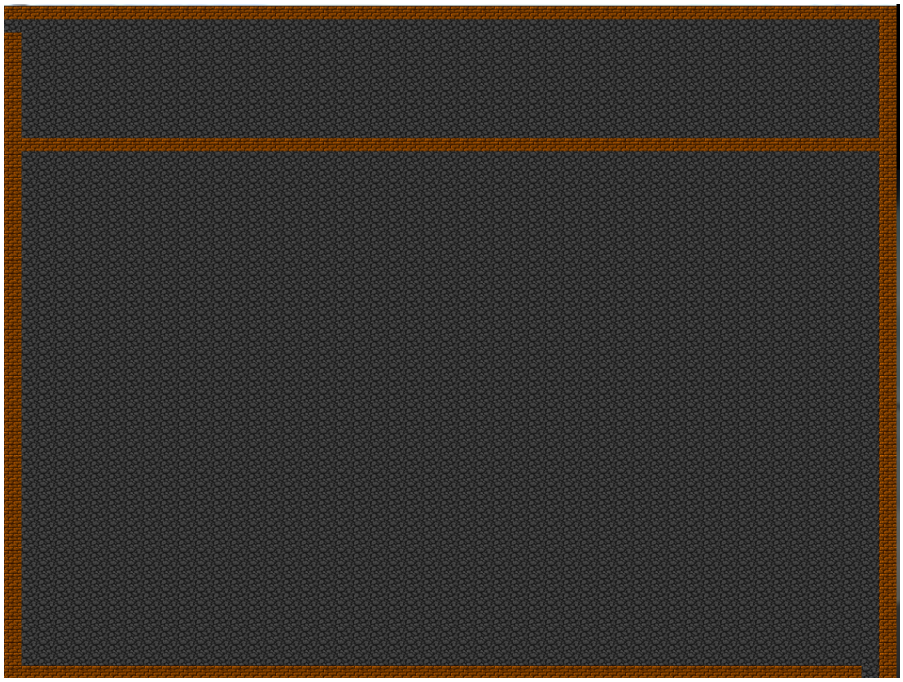
(The following images use the same algorithm implemented in another programming language so that images could be attached to the maze for display purposes)

The maze generation algorithm broken down:

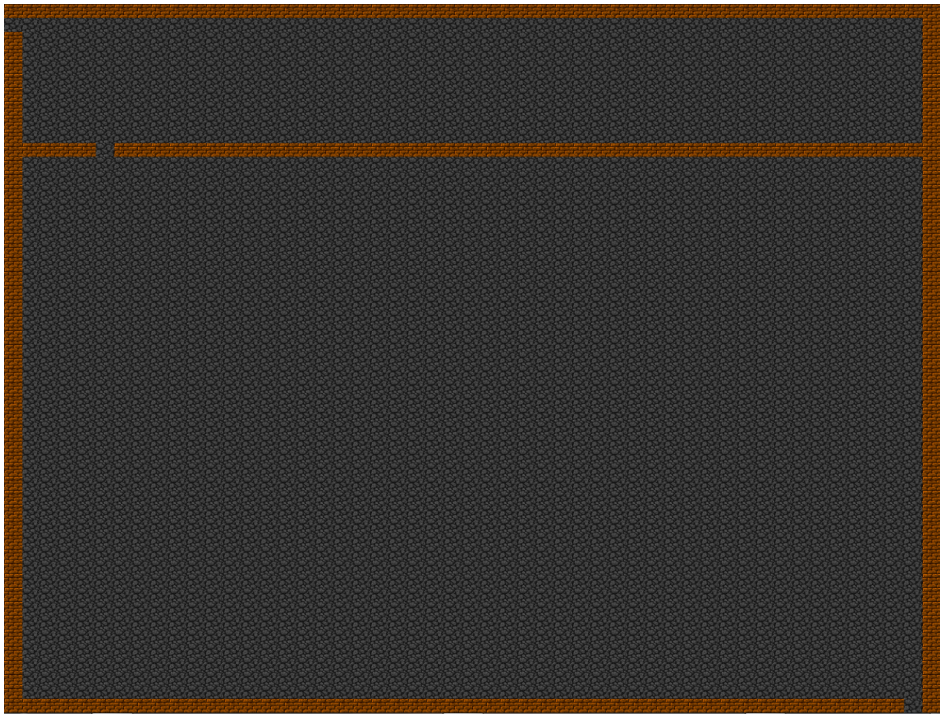
Step 1: The array is initialized to contain all walkways, except for a perimeter of walls.

Alex RInehart
Ben Myshkowec
Group E3

Step 2: An entrance and exit are randomly positioned (Each is in one of two positions, near the upper left and lower right corners, respectively).

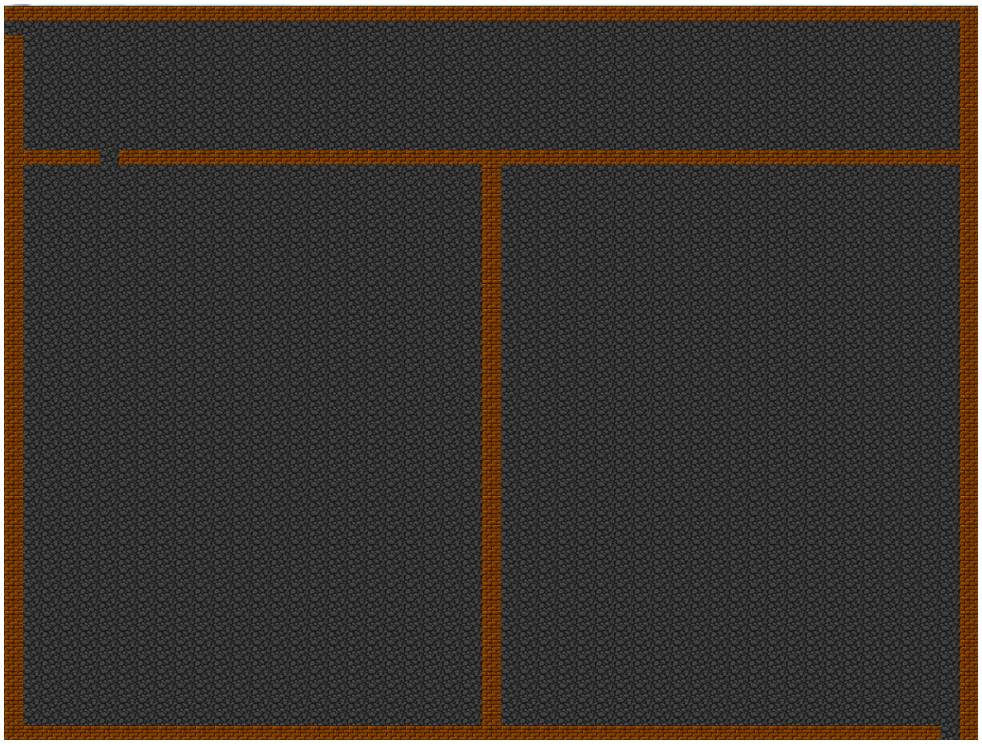


Step 3: A partition is randomly selected. The direction is based on the dimensions of the current array.



Alex RInehart
Ben Myshkowec
Group E3

Step 4: A walkways is created in the partition created above. Its position is random, but within the wall. This guarantees that each segment will be connected to the previous. The algorithm makes careful use of even/odd positioning to ensure that a vertical split of a section cannot run through a previously created walkway. (Walkways are only on even numbered areas, splits are only on odd).



Step 5. The process is repeated recursively on every new sub-array created until the section is smaller than the chosen resolution. The resolution is the maximum size of walkway, or the space between walls.

A subarray is defined as the left, right, top, and bottom corners. This is what is passed to the generate and split functions.