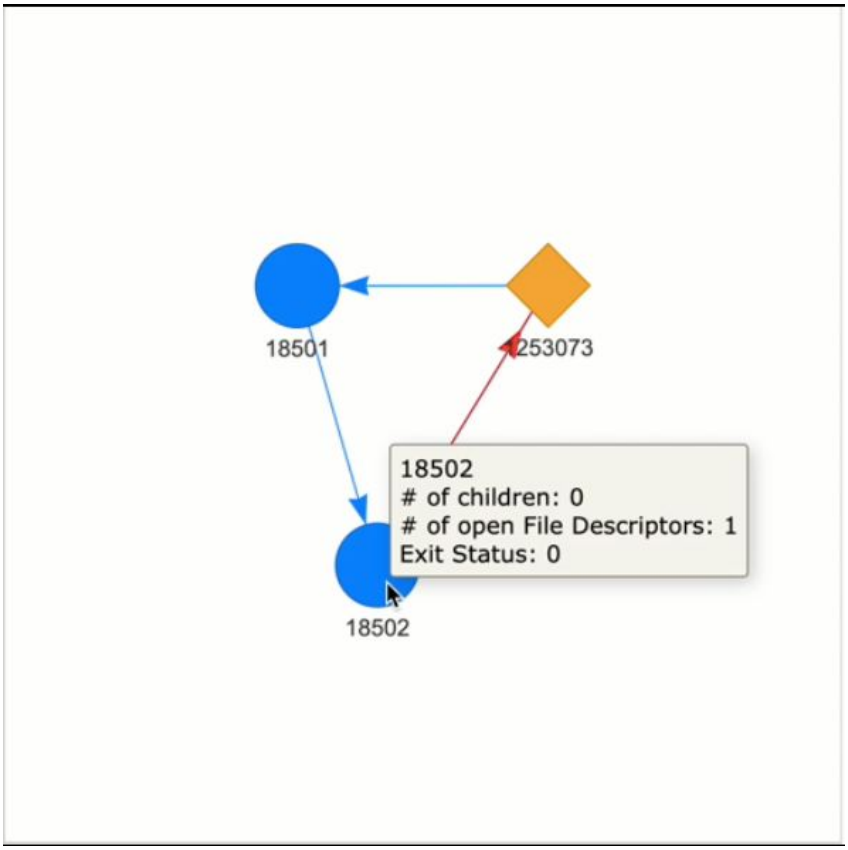
About the debugger:

This debugger allows you to run a C-program, and visualize all the parent/child relationships formed in it as well as the communication that occurs between processes (using files/pipes).

How to run it:

- cd into the folder in which you have the pdt binary file and pdt.py
- Run python3 pdt.py in this folder
- When this program takes input for a program to run - write the name of the program you wish to visualize
- After pdt.py has finished running, you will see 2 HTML files {your program's name}.html and {your program's name}_gannt_chart.html. The former will show you a diagram that depicts the relationships between processes (using circles for processes and blue arrows to represent forking) and communication between them (with squares for files or inodes that were read or written to). For example:

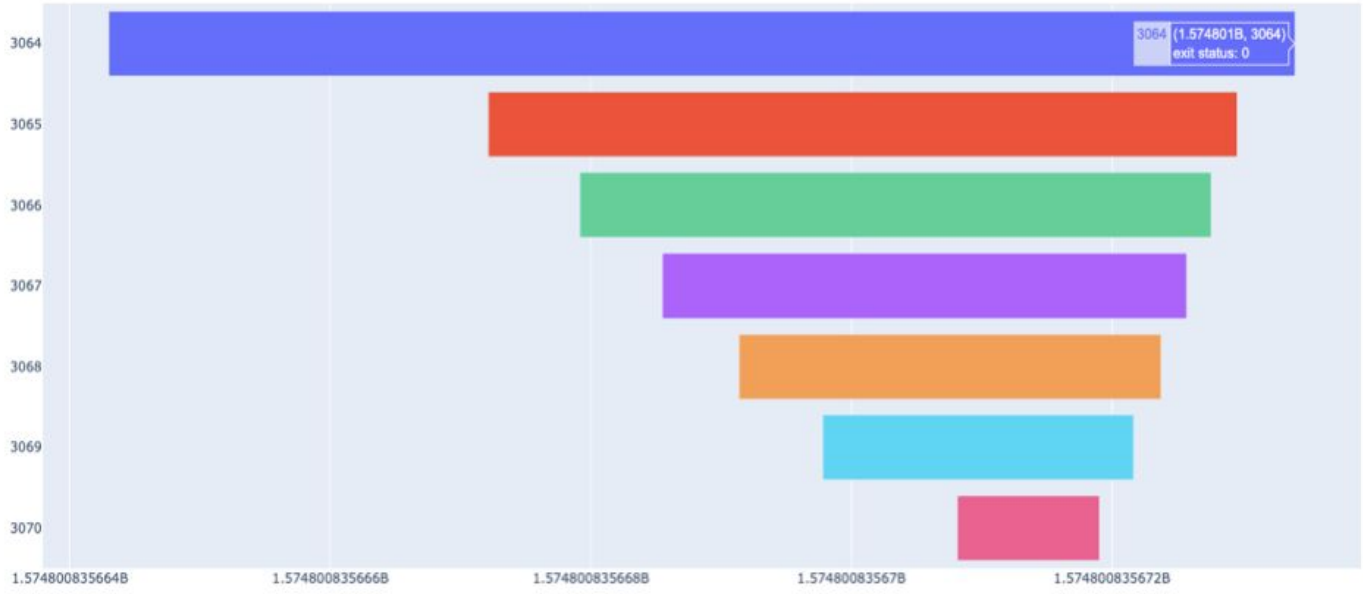


In the graph above, we can see that:

- Process 18501 forked process 18502
- Process 18502 wrote something to inode 1253073 (you can hover over the arrow from 18502 to 125073 or inode square to see what it wrote)

        - Process 18501 read something from inode 1253073 (you can hover over the arrow from the 125073 to 18501 to see what it read or hover over the inode square itself)
        - Process 18502 forgot to close the file descriptor which it used to write to inode 1253073 (hence, it is read)
        - Process 18502 has 0 children, 1 open file descriptor (shown by the red arrow) and exit status 0. To see this information, we simply hovered over the process node.

- When you open the gannt chart file, you will see a plot that depicts the execution time for each process. This can help see when a process started and exited relative to other processes. You can hover over the bars in this gannt chart to see the PID of the process whose execution time you are seeing, as well as the process's exit status. To see the relationship between the processes, however, you will need to see the process/files graph discussed above.
    - Example:



Implementation:

- Tracing:

    - To trace programs, we used the ptrace system call in *pdt.c*. This system call allows using a process (tracer) to trace another process (tracee) getting attached to it. Once attached, the tracer waits for the tracee to stop with a sigtrap signal every time since the tracee stops in this manner every time it makes a syscall. At this point, the tracer can peek into the tracee's memory registers (and change them) to see what system call is made, what arguments it provided and what the return status was before letting the tracee resume running again.
    - For our implementation, we used this ptrace feature to keep track of read, write, open, close, pipe, fork and exit system calls
    - We used an AVL tree to keep track of the processes being traced (since we trace multiple processes). Each node in this AVL tree kept track of a process's PID, children, and open fds. We could've chosen other data structures to keep track of the processes too (like linked lists) but we chose an AVL tree since we were frequently inserting, searching and deleting (when a process exited). These operations take O(logn) time with AVL trees which was better than the other options we were considering at that time.
    - We used linked lists to keep track of what processes wrote/read (storing the PID, the inode of the file written to/read from, and information read/written) and what processes exited. We also used linked lists to keep track of children and open files for each process being traced.
    - When there are no remaining processes left to trace, we write all the data collected to a test.csv file which is parsed and visualized in Python

- Parsing and Plotting:
    - We parse the csv file generated by *pdt.c* and store it in 3 dictionaries which are used to generate a process graph and gannt chart
    - To generate the process graph, we use Pyvis.
    - To generate the gannt chart, we use Pandas and Plotly

Difficulties

- Learning how to use Ptrace appropriately:
    - We experienced a lot of difficulties when trying to understand which system calls were valid. Since we didn't know that a system call is valid only if the status which the child stopped with is SIGTRAP, we spent a lot of time confused about the weird pattern of system calls we were saying. For instance, we thought some processes were writing after they had exited. However, we realized later that the status the child has stopped with wasn't a SIGTRAP signal, so we were just reading some left-over values. We also found out that some processes randomly stopped due to SIGCHLD signals. This was also leading to some misleading results since the process had not made a system call

- Downloading the required Python libraries on lab computers took quite some time, so we had to repeatedly scp the test.csv file made by pdt.c on the lab pcs to our native computers, and run the python script on this csv file
- Filtering for the correct system calls- A huge part of the project was figuring out which systems calls we would access data from and which we would ignore. There were some cases where signals such as SIGCHLD interfere with the way our program worked, causing discrepancies in the data collected.

Directory Structure:

The following files (in our repository) are required for this debugger:

- pdt.h and pdt.c: all of our program tracing is done in pdt.c, while pdt.h is the header file for pdt.c (includes all our function declarations and documentation)
- avl_tree.h and avl_tree.c: The avl struct we use to keep track of each process and linked lists to keep track of its children and open file descriptors are in avl_tree.h (with descriptions). Helpers we needed to read these structures/modify them are in avl_tree.c (with declarations in avl_tree.h)
- dead_ll.h and dead_ll.c: Linked list and helpers for storing data about processes that exited
- log.h and log.c: Linked list to keep track of read/write calls (each node stores PID that made read/write call, inode that was written to/read from, and data that was written/read)
- Makefile: helps compile the files above into one binary file, ./pdt which can be used to generate program traces, and add them to a csv file (test.csv)
- pdt.py: Python program that calls ./pdt to generate traces for a program and uses these to make a process graph and gannt chart (as described above)

Testing Suite:

Blank- Empty file, Tests trivial case

Segfault- code segfaults, Tests Segfaults

Simple_fork - small multi-process example, Tests forking with one child

twoPipe - a two-way interprocess communication with 2 levels of forking, Tests pipes, multiple-reads,multiple-writes, one chain of process forking

presentationForloop - forks in a for loop, Tests multiple children, multiple forking

onePrint - prints one line to stdout, Tests to make sure STDOUT and STDIN's data is not read from

Pfact - using the sieve of eratosthenes algorithm to filter prime numbers, Test large case of chain of multi processes with multiple read and write calls, also tests to make sure numbers are stored in hex.

Multiple_siblings - a giant chain of processes (similar to presentationForLoop) but one child segfaults, Tests segfaults, multiprocess forking and to make sure that all data from the non segfaulting processes is collected before the program terminates.

presentationMultipipe - creates 4 processes and adds a pair of pipes between them, Tests multi process forking and piping.