pi contains an array of portinfo(which is used to hold info for a particular port). This array is initialised in the shared memory.

Misc: {
usec_delay.it_interval.tv_usec:

- usec_delay is a variable of type struct itimerval. This is used to store if the micro second delay option is set.
- it_interval is a field within the struct itimerval structure, representing the interval between timer expirations.
- tv_usec is a field within the struct timeval structure, representing the number of microseconds within the time interval.

}

The Scanmain function is the main function for the scanning mode of the program. It first initializes a shared memory segment to store information about the ports to be scanned. This shared memory segment is represented by the pi variable, which is an array of portinfo structures. Each portinfo structure represents a port and contains information about whether the port is active, the number of retries left for the port, and the time the port was last sent a probe.

The function then parses the ports to be scanned from the opt_scanports string and stores the information in the pi array. It also counts the total number of ports to be scanned and prints this information.

After setting up the shared memory and parsing the ports, the function forks a new process. The parent process becomes the receiver, which receives the responses to the probes sent to the ports. The child process becomes the sender, which sends the probes to the ports.

If the program receives a SIGINT, or SIGTERM signal, it will exit. Both the process handles these signals by calling the do_exit function. There is a special signal SIGCHLD, send when the child completes it sends this signal to the parent process. The sender function will terminate after all ports have been scanned, which will send the signal and parent process will respond by calling do_exit.

Parse_ports(): The parse_ports function is used to parse the ports to be scanned from an input string and store the information in an array of *portinfo* structures. The string is expected to contain port numbers or ranges separated by commas, it also supports **port negation** i.e we can mention the ports that shouldn't be scanned.

The function starts by tokenizing the string into an array of strings, where each string represents a port or a range of ports. It then iterates over each string in the array. The function used is strftok which is customized for this particular use case but built on top of strtok.

If a string starts with a '!', it indicates that the ports specified in the string should not be scanned. The function sets the *neg* variable to 1 and removes the '!' from the string.

All the string comparisons are done using strcmp function.

If a string contains a '-', it indicates a range of ports. The function tokenizes the string into two strings representing the lower and upper bounds of the range. It then sets the *active* field of the *portinfo* structures for all ports in the range to *!neg*. Also, at every point it does syntax checks i.e looking for syntax errors.

If a string is "all", the function sets the *active* field of the *portinfo* structures for all ports to *!neg*.

If a string is "known", the function sets the active field of the *portinfo* structures for all ports listed in the /etc/services file to *!neg*.

If a string is a single number, the function sets the active field of the *portinfo* structure for the port to *!neg*.

If the function encounters a syntax error while parsing the string, it returns 1. Otherwise, it returns 0.

Sender(): This function is run by the child process after the scanmain function forks. The function continues to loop until all ports have been scanned, i.e., all ports are inactive.

Working:

1. A variable *retry* is initialized to 0 which keeps a track of how many times scanning loop has run. Then, it enters a loop that continues until all ports have been scanned. Inside the loop, it iterates over each port in the shared memory structure *pi*. For each port, it checks if the port is active and if any retries are left for that port. (initially the *retry* value set to *opt_scan_probes* = 8). If it is active and has retries left then *active* variable incremented which keeps a count of active ports in that scanning iteration.

2. If the port is active and has retries left, it sends a probe packet to that port by calling the *send_tcp* function.

3. After sending the probe *sleep* if the microsecond(*opt_waitinusec*) wait is set then *wait* for (*usec_delay.it_interval.tv_usec*) i.e refer at top, otherwise for *sending_wait* (set to 1s). It then decreases the retry count for that port and increases the total number of active ports, also sequence (denoting sequence number of packets) is reset to -1 and destination port is set to i. Sent time is also recorded in the pi[i].sentms (in milliseconds).

4. avrgms <- rtt value which is stored in *pi[MAXPORT + 1].active*; Once more than 3 iterations of the complete scanning loop has occurred, print it if in debug mode, else if it is non-zero sleep it for that time, done using usleep(by multiplying 1000 to make it milliseconds), otherwise sleep it for 1 sec. This is done to make sure all the packets are

received.

5. Now, we check the number of packets received, once received the port is made inactive but it would still have atleast one retry left so we check that to count the number of *recvd* (recvd packets).

6. Once, all the retries of active ports have been exhausted (that means the *active* variable will be 0) we can terminate the loop, but if no packets received at all then still sleep just to make sure no packets are missed. The non-responding ports are displayed and exited.

7. (This occurs only when microsecond wait selected)Else we need to check if are going too fast or not if so we would need to tweak the values of microsecond wait time.

Receiver(): The *receiver* function is responsible for receiving and processing the responses to the packets sent by the *sender* function. It continuously reads packets and processes them based on their type and content.

Here is a brief overview of how the receiver function works:

1. It enters an infinite loop where it continuously reads packets using the read_packet function.

2. It performs some basic sanity checks on the received packet, such as checking if the length of the packet is greater than or equal to the size of the link header and if the length of the IP part of the packet is greater than or equal to the size of the IP header.

3. It checks if the protocol is not ICMP and if the destination IP address in the packet matches the local IP address and if the source IP address matches the remote IP address. Otherwise, throw error. The comparisons occur using memcmp.

4. If the protocol of the packet is TCP, it performs additional checks and processing. It checks if the destination port in the TCP header matches the initial source port and if the source port is active. If these conditions are met, it calculates the round trip time (RTT) for the packet, updates the average RTT, and prints information about the packet and marks the port inactive. In case the conditions aren't met the loop continues.

5. If the protocol of the packet is ICMP, it performs additional checks and processing. It checks if the quoted packet in the ICMP packet matches the original packet sent by the sender function which includes source and destination ip checks mentioned above, it also checks if the quoted tcp packet port matches, it also checks if the mentioned source port is active. If this condition is met, it prints information about the packet, otherwise it continues looping.

6. The function continues to loop, reading and processing packets, until the program is terminated.