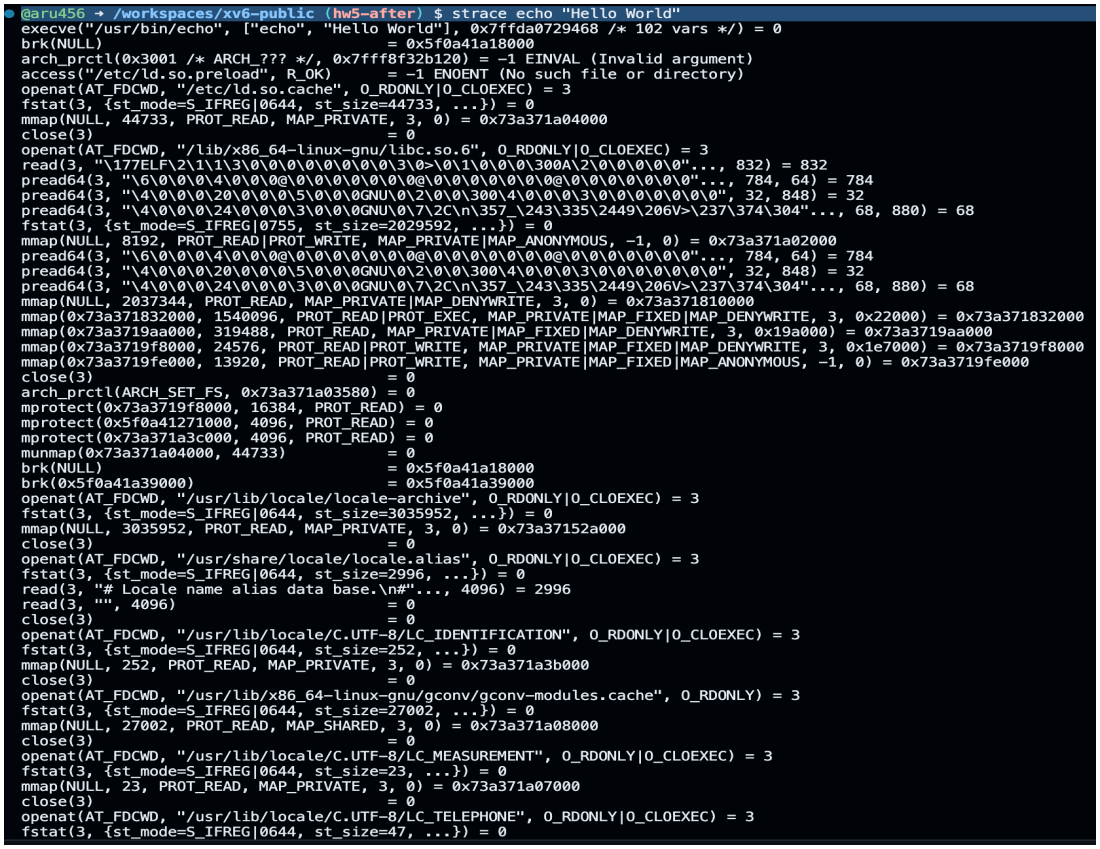
Tasks:

4.1 Get familiar with Linux strace command

4.1.1 Use strace in Linux

Strace is a powerful Linux tool that reveals how a program interacts with the system by showing the system calls it uses. For example, running `strace echo hello` lets you see the internal actions as it processes and displays "hello," making it useful for debugging and understanding program behavior.



```
openat(AT_FDCWD, "/usr/lib/locale/C.UTF-8/LC_TELEPHONE", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=47, ...}) = 0
mmap(NULL, 47, PROT_READ, MAP_PRIVATE, 3, 0) = 0x73a371a06000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/locale/C.UTF-8/LC_ADDRESS", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=131, ...}) = 0
mmap(NULL, 131, PROT_READ, MAP_PRIVATE, 3, 0) = 0x73a371a05000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/locale/C.UTF-8/LC_NAME", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=62, ...}) = 0
mmap(NULL, 62, PROT_READ, MAP_PRIVATE, 3, 0) = 0x73a371a04000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/locale/C.UTF-8/LC_PAPER", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=34, ...}) = 0
mmap(NULL, 34, PROT_READ, MAP_PRIVATE, 3, 0) = 0x73a371529000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/locale/C.UTF-8/LC_MESSAGES", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/locale/C.UTF-8/LC_MESSAGES/SYS_LC_MESSAGES", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=48, ...}) = 0
mmap(NULL, 48, PROT_READ, MAP_PRIVATE, 3, 0) = 0x73a371528000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/locale/C.UTF-8/LC_MONETARY", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=270, ...}) = 0
mmap(NULL, 270, PROT_READ, MAP_PRIVATE, 3, 0) = 0x73a371527000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/locale/C.UTF-8/LC_COLLATE", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=1518110, ...}) = 0
mmap(NULL, 1518110, PROT_READ, MAP_PRIVATE, 3, 0) = 0x73a3713b4000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/locale/C.UTF-8/LC_TIME", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=3360, ...}) = 0
mmap(NULL, 3360, PROT_READ, MAP_PRIVATE, 3, 0) = 0x73a3713b3000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/locale/C.UTF-8/LC_NUMERIC", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=50, ...}) = 0
mmap(NULL, 50, PROT_READ, MAP_PRIVATE, 3, 0) = 0x73a3713b2000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/locale/C.UTF-8/LC_CTYPE", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=201272, ...}) = 0
mmap(NULL, 201272, PROT_READ, MAP_PRIVATE, 3, 0) = 0x73a371380000
close(3)                                = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
write(1, "Hello World\n", 12Hello World
)           = 12
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

```
@aru456 ➜ /workspaces/xv6-public (hw5-after) $ strace -c echo "Hello World"
Hello World
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 21.01    0.000291          13        21           mmap
 19.06    0.000264         264         1           execve
 18.48    0.000256          14        18           openat
 13.79    0.000191           9        20           close
 12.13    0.000168           8        19           fstat
  5.49    0.000076          12         6           pread64
  2.60    0.000036          12         3           mprotect
  1.95    0.000027           9         3           read
  1.59    0.000022           7         3           brk
  1.23    0.000017          17         1           munmap
  1.08    0.000015          15         1           write
  0.87    0.000012           6         2         1 arch_prctl
  0.72    0.000010          10         1         1 access
------ ----------- ----------- --------- --------- ----------------
100.00    0.001385                    99         2 total
@aru456 ➜ /workspaces/xv6-public (hw5-after) $
```

4.1.2 Four System Calls

i. **execve**
   - The execve system call is used to execute a program. It replaces the current process image with a new one.
   - In the echo "Hello World!" command, execve is used to launch the echo program located at /usr/bin/echo, with the arguments "echo" and "Hello World!". It also passes the current environment variables to the new process.

ii. **openat**
   - The `openat` system call opens files relative to a directory file descriptor. It is often used to open libraries, configuration files, or system resources.
   - During the execution of `echo`, `openat` is used to open shared libraries and locale files required for the program to run. For example, it opens `/etc/ld.so.cache` to locate shared libraries.

iii. **write**
   - The `write` system call writes data to a file descriptor. A file descriptor could refer to standard output (`stdout`), standard error (`stderr`), or an actual file.
   - In the `echo` command, `write` is used to write the string `"Hello World!\n"` to the terminal, which corresponds to file descriptor `1` (standard output).

iv. **close**
   - The `close` system call closes a file descriptor, ensuring that resources associated with the file are released.
   - After `echo` finishes accessing files (e.g., shared libraries or locale files), it uses `close` to release the file descriptors and free resources.

4.2 Building strace in xv6

4.2.1 Implement “strace on”:

sysproc.c

```
int
sys_strace(void)
{
    int mode;
    if (argint(0, &mode) < 0)
        return -1;

    struct proc *curproc = myproc();
    curproc->tracing = mode; // Enable or disable tracing


    return 0;
}
```

In this code, we implemented the `sys_strace` system call to enable or disable system call tracing for the current process. The `argint` function retrieves the mode (1 for enabling, 0 for disabling), and the `curproc->tracing` flag is updated accordingly to control tracing behavior for this process.

syscall.c

```
void
syscall(void)
{
    int num;
    struct proc *curproc = myproc();

    num = curproc->tf->eax; // System call number
    if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        int retval = syscalls[num](); // Execute the system call
        curproc->tf->eax = retval;

        // Log system call if tracing is enabled
        if (curproc->tracing) {
            cprintf("TRACE: pid = %d | command_name = %s | syscall = %s | return value = %d\n",
                    curproc->pid, curproc->name, syscallnames[num], retval);
            add_event(curproc->pid, curproc->name, syscallnames[num], retval);

        }
    } else {
        cprintf("%d %s: unknown sys call %d\n",
                curproc->pid, curproc->name, num);
        curproc->tf->eax = -1;
    }
}
```

In this `syscall` function, we handle system calls for the current process. It retrieves the system call number, validates it, and executes the corresponding function. If tracing is enabled (`curproc->tracing`), it logs details like the process ID, command name, system call name and return value. If the system call number is invalid, it logs an error and returns `-1`.

proc.h

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int tracing;  // Tracing flag: 1 for on, 0 for off
};
```

We added a `tracing` field in the `proc` structure to act as a flag for enabling or disabling system call tracing for each process.

```
$ echo hello
hello
$ strace on
TRACE: pid = 4 | command_name = strace | syscall = strace | return value = 0
$ echo hello
TRACE: pid = 5 | command_name = sh | syscall = sbrk | return value = 16384
TRACE: pid = 5 | command_name = echo | syscall = exec | return value = 0
hTRACE: pid = 5 | command_name = echo | syscall = write | return value = 1
eTRACE: pid = 5 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 5 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 5 | command_name = echo | syscall = write | return value = 1
oTRACE: pid = 5 | command_name = echo | syscall = write | return value = 1

TRACE: pid = 5 | command_name = echo | syscall = write | return value = 1
$
```

The output of `strace on` and `echo hello` is generated by combining functionality across the modified files. The `sys_strace` system call sets the `tracing` flag in the `proc` structure, enabling system call logging for the current process and its descendants. The `syscall` function, responsible for handling all system calls, checks this flag and logs the `pid`, `command_name`, `syscall`, and `return value` using the `syscallnames` array to map system call numbers to their names. The `proc` structure provides process-specific details, such as the name (`echo`) and tracing status, to ensure accurate and meaningful logs. This coordination enables `strace` to produce the detailed system call logs seen in the output.

4.2.2 Strace off

strace.c

```
if (strcmp(argv[1], "on") == 0) {
  strace(1); // Enable tracing
} else if (strcmp(argv[1], "off") == 0) {
  strace(0); // Disable tracing
```

This code enables or disables system call tracing based on the argument provided (on or `off`) by calling the `strace` function with `1` to enable or `0` to disable tracing.

OUTPUT:

This output shows that `strace` was disabled (`strace off`), but the last few traced system calls, such as `sbrk` and `exec`, were logged before the tracing was turned off. Subsequent commands like `echo hello` are no longer traced

```
$ strace off
TRACE: pid = 6 | command_name = sh | syscall = sbrk | return value = 16384
TRACE: pid = 6 | command_name = strace | syscall = exec | return value = 0
$ echo hello
hello
$
```

4.2.3 Strace run <command>

strace.c

```
else if (strcmp(argv[1], "run") == 0) {
  if (argc < 3) {
      printf(2, "Usage: strace run <command> [args]\n");
      exit();
  }

  // Fork a new process
  int pid = fork();
  if (pid < 0) {
      printf(2, "Fork failed\n");
      exit();
  } else if (pid == 0) {
      // In child process: enable tracing for the child process
      strace(1);

      // Execute the command
      exec(argv[2], &argv[2]);
      printf(2, "exec %s failed\n", argv[2]);
      exit();
  } else {
      // In parent process: wait for child to finish
      wait();
  }
```

This code implements the `strace run` functionality, which forks a new process to execute the specified command with system call tracing enabled. The parent process waits for the

child to complete, while the child process enables tracing using `strace(1)` and executes the provided command using `exec`. If the `fork` or `exec` calls fail, appropriate error messages are displayed.

OUTPUT:

When `strace run echo hello` is executed, it tracks all the system calls made by the `echo` command. The logs show `exec` being called to start the `echo` process, followed by multiple `write` system calls as each part of "hello" is printed to the terminal. Each entry includes the process ID, the command name, the system call being made, and its result, offering a clear view of how `echo` interacts with the system.

```
$ strace run echo hello
TRACE: pid = 8 | command_name = strace | syscall = strace | return value = 0
TRACE: pid = 8 | command_name = echo | syscall = exec | return value = 0
hTRACE: pid = 8 | command_name = echo | syscall = write | return value = 1
eTRACE: pid = 8 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 8 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 8 | command_name = echo | syscall = write | return value = 1
oTRACE: pid = 8 | command_name = echo | syscall = write | return value = 1

TRACE: pid = 8 | command_name = echo | syscall = write | return value = 1
$
```

4.2.4 Strace dump

sysproc.c

```
int
sys_stracedump(void) {
  struct event *current = event_list;

  if (!current) {
    cprintf("No strace events to dump.\n");
    return 0;
  }

  while (current) {
    cprintf("TRACE: pid = %d | command = %s | syscall = %s | return value = %d\n",
            current->pid, current->name, current->syscall, current->ret);
    current = current->next;
  }

  return 0;
}
```

This function, `sys_stracedump`, iterates through a list of recorded system call events and prints their details, including the process ID, command name, system call, and return value. If no events are available, it displays a message indicating there are no trace events to dump.

Syscall.h

```
23    #define SYS_strace 22
24    #define SYS_stracedump 23
25
```

Defined SYS_strace and SYS_stracedump in syscall.h

Syscall.c

```c
struct event *event_list = 0; // Initialize the linked list head
int event_count = 0;          // Initialize the event counter

void add_event(int pid, const char *name, const char *syscall, int ret) {
  struct event *new_event = (struct event *)kalloc();
  if (!new_event) {
    cprintf("Failed to allocate memory for event\n");
    return;
  }

  // Populate the event details
  new_event->pid = pid;
  safestrcpy(new_event->name, name, sizeof(new_event->name) - 1);
  safestrcpy(new_event->syscall, syscall, sizeof(new_event->syscall) - 1);
  new_event->ret = ret;
  new_event->next = event_list;

  // Update the linked list
  event_list = new_event;
  event_count++;

  // Remove the oldest event if the list size exceeds N
  if (event_count > N) {
    struct event *current = event_list;
    while (current->next && current->next->next) {
      current = current->next;
    }

    kfree((char *)current->next);
    current->next = 0;
    event_count--;
  }
}
```

This code keeps a record of system call events using a linked list. Each event stores details like the process ID, command name, system call, and return value. New events are added to the top of the list, and if the list grows too large, the oldest events are removed to make space, ensuring the log stays within the defined limit.

Syscall.c

```c
void
syscall(void)
{
    int num;
    struct proc *curproc = myproc();

    num = curproc->tf->eax; // System call number
    if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        int retval = syscalls[num](); // Execute the system call
        curproc->tf->eax = retval;

        // Log system call if tracing is enabled
        if (curproc->tracing) {
            cprintf("TRACE: pid = %d | command_name = %s | syscall = %s | return value = %d\n",
                    curproc->pid, curproc->name, syscallnames[num], retval);
            add_event(curproc->pid, curproc->name, syscallnames[num], retval);

        }
    } else {
        cprintf("%d %s: unknown sys call %d\n",
                curproc->pid, curproc->name, num);
        curproc->tf->eax = -1;
    }
}
```

This `syscall` function handles system calls for the current process, logging details like process ID, command name, system call, and return value if tracing is enabled. If the system call is invalid, it logs an error and returns `-1`.

strace.c

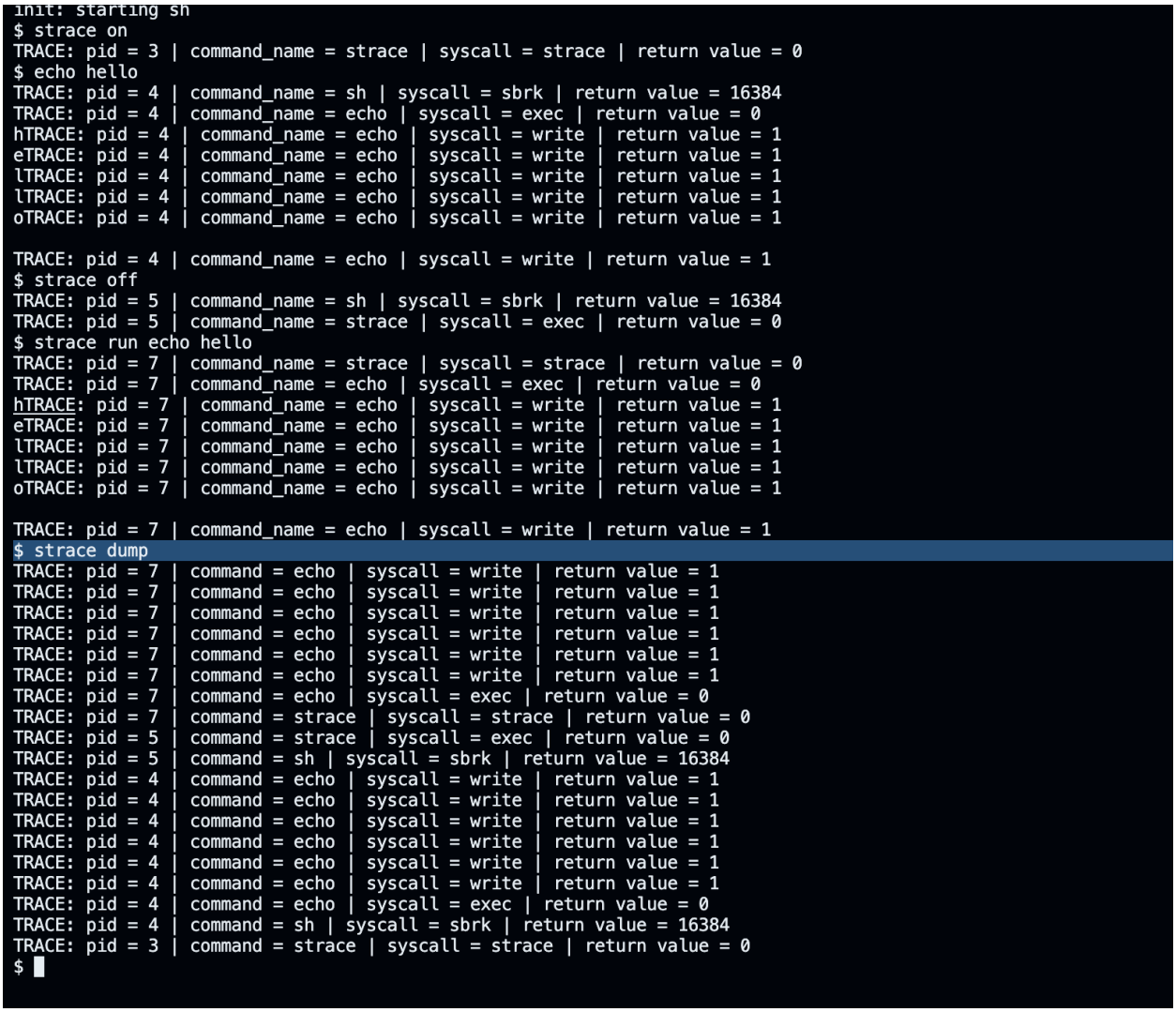
```c
} else if (strcmp(argv[1], "dump") == 0) {
  int ret = stracedump();
  if (ret < 0) {
    printf(2, "Failed to dump trace events\n");
  }
```

This code calls the `stracedump` function to display all logged system call events and prints an error message if the dump fails.

proc.h

```c
#define N 100 // Maximum number of events to store

struct event {
  int pid;
  char name[16];
  char syscall[16];
  int ret;
  struct event *next;
};
```

This `struct event` defines a linked list node to store details of a traced system call, including the process ID, command name, system call name, return value, and a pointer to the next event.



strace dump - OUTPUT Explanation:

The `strace dump` command outputs all logged system call events, including details like process ID, command name, system call, and return value. It shows a comprehensive trace of all previously executed commands, such as `echo` and `strace`, listing their system calls along with their results. This provides a complete history of traced system calls for debugging or analysis.

4.2.5 - Trace child process

proc.h | strace_test.c

strace_test.c > ...

```c
#include "types.h"
#include "user.h"

int main() {
    printf(1, "Starting trace test program\n");

    // Enable tracing
    strace(1);

    int pid = fork();
    if (pid < 0) {
        printf(2, "Fork failed\n");
    } else if (pid == 0) {
        // In child process
        printf(1, "In child process: PID = %d\n", getpid());
        printf(1, "Child process running echo\n");
        char *args[] = {"echo", "hello from child", 0};
        exec("echo", args);
        printf(2, "Exec failed\n");
    } else {
        // In parent process
        wait(); // Wait for the child process to finish
        printf(1, "In parent process: PID = %d\n", getpid());
    }

    // Dump trace
    printf(1, "Dumping trace events:\n");
    stracedump();

    exit();
}

```

This test program enables system call tracing, forks a child process, and runs the `echo` command in the child. The parent process waits for the child to finish, and after execution, it dumps the traced events using `stracedump`. It demonstrates tracing across parent and child processes.

OUTPUT:

The output shows the parent process creating a child process to run the `echo` command. The child process starts `echo` and logs its actions, like writing the text to the screen. Meanwhile, the parent process waits for the child to finish. These logs give a clear view of how the parent and child processes work together and interact with the system.

```
$ strace_test
Starting trace test program
TRACE: pid = 9 | command_name = strace_test | syscall = strace | return value = 0
TRACE: pid = 9 | command_name = strace_test | syscall = fork | return value = 10
TRACE: pid = 10 | command_name = strace_test | syscall = getpid | return value = 10
ITRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
nTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
cTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
hTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
iTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
lTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
dTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
pTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
rTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
oTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
cTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
eTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
sTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
sTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
:TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
PTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
ITRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
DTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
=TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
1TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
0TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
```

```
TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
CTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
hTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
iTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
lTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
dTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
pTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
rTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
oTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
cTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
eTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
sTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
sTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
rTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
uTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
nTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
nTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
iTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
nTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
gTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
eTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
cTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
hTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
oTRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
```

```
TRACE: pid = 10 | command_name = strace_test | syscall = write | return value = 1
TRACE: pid = 10 | command_name = echo | syscall = exec | return value = 0
hTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
eTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
oTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
 TRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
fTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
rTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
oTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
mTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
 TRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
cTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
hTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
iTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
dTRACE: pid = 10 | command_name = echo | syscall = write | return value = 1

TRACE: pid = 10 | command_name = echo | syscall = write | return value = 1
TRACE: pid = 9 | command_name = strace_test | syscall = wait | return value = 10
TRACE: pid = 9 | command_name = strace_test | syscall = getpid | return value = 9
ITRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
nTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
pTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
aTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
rTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
eTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
nTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
tTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
pTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
rTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
oTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
cTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
eTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
sTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
sTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
:TRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
PTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
ITRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
DTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
=TRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
9TRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
```

```
TRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
DTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
uTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
mTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
pTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
iTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
nTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
gTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
tTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
rTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
aTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
cTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
eTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
 TRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
eTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
vTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
eTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
nTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
tTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
sTRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
:TRACE: pid = 9 | command_name = strace_test | syscall = write | return value = 1
```

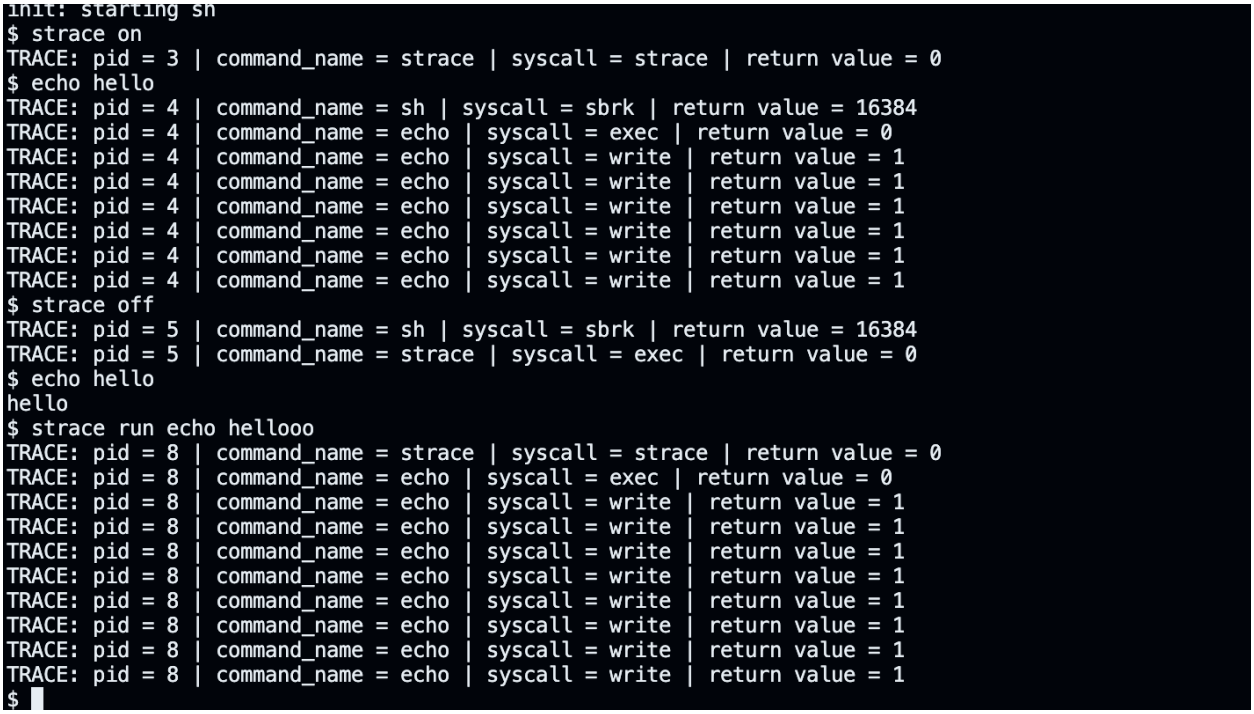
4.2.6: Extra credits: Formatting more readable output

```c
int
sys_write(void)
{
    struct file *f;
    int n;
    char *p;
    struct proc *curproc = myproc();

    if (argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
        return -1;

    // If tracing is enabled, suppress command output
    if (curproc->tracing) {
        if (f->writable == 0)
            return -1; // Non-writable file, return error
        else
            return n; // Simulate a successful write
    }

    // Default behavior for normal writes
    return filewrite(f, p, n);
}
```

In this code, we added a check to see if `strace` is turned on for the process. If tracing is active, the normal output from commands is suppressed. For files that can't be written to, it simulates a write error, and for others, it skips the actual writing while pretending it succeeded. If tracing is off, everything works normally, and the command outputs are displayed as usual. This change ensures that only the `strace` logs are visible when tracing is enabled, hiding any regular command output. So from this point the norma output cannot be seen in the code

OUTPUT:

This output demonstrates how `strace` captures system calls when tracing is turned on. At first, you can see both the system calls and the output of commands like `echo hello`. However, when `strace run` is used, the suppression feature activates, hiding the actual output (`hellooo`) and logging only the system calls, such as `write`. This shows that the suppression is working correctly, ensuring that only the trace logs are displayed during active tracing.



4.3: Building options for strace

- ### 4.3.1: Option: -e <system_call_name> [5 points]

`-e` (Filter Specific System Calls): The `-e` option focuses on specific system calls. For example, using `strace -e write` will only show the `write` system calls while skipping others. This helps in understanding how a particular system call is used in the program.

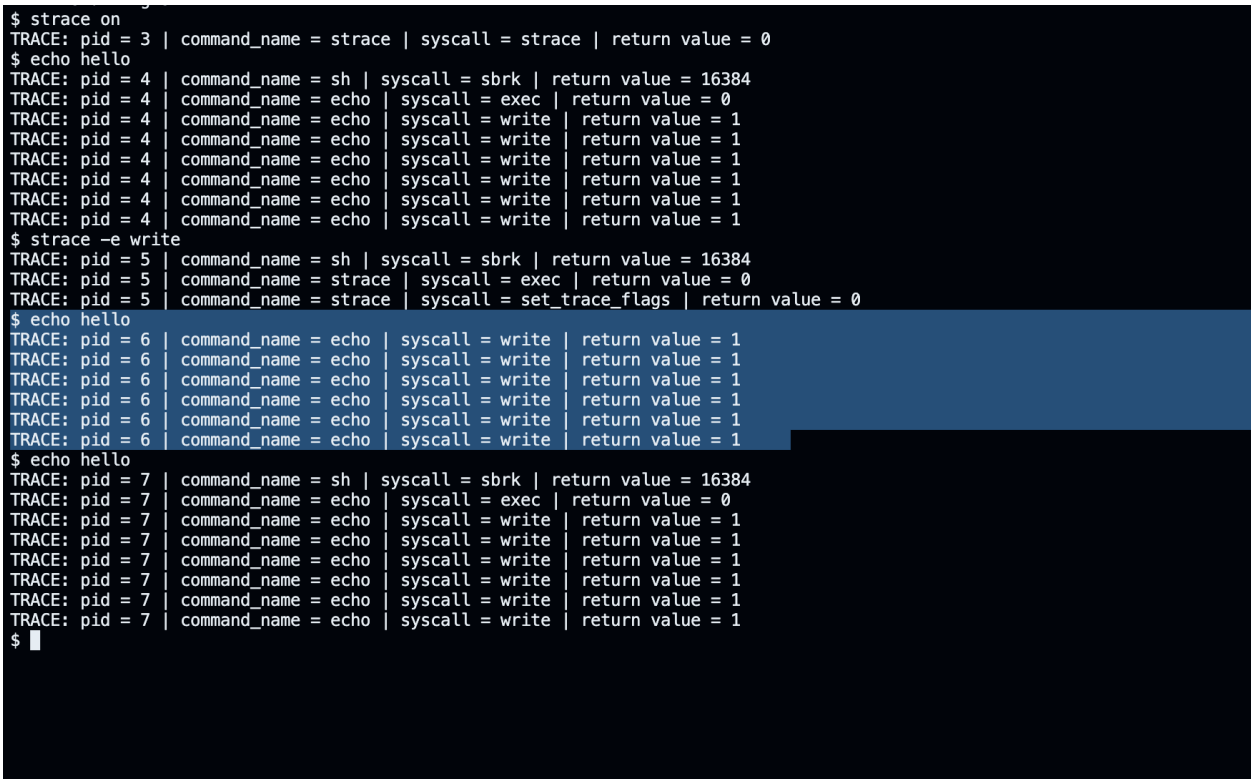
We added a feature to allow tracing specific processes by setting filters for system calls and flags to log only successful or failed calls. The `sys_set_trace_flags` function prepares these settings for the next process that gets spawned, and the `reset_trace_flags` function clears them once that process finishes. This way, we can focus on targeted tracing while ensuring everything resets cleanly afterward.

Sysproc.c

```
int sys_set_trace_flags(void) {
    char *filter;
    int success_flag, failure_flag;

    if (argstr(0, &filter) < 0 ||
        argint(1, &success_flag) < 0 ||
        argint(2, &failure_flag) < 0) {
        return -1;
    }

    struct proc *curproc = myproc();

    // Configure tracing for the next process (PID + 1)
    safestrcpy(curproc->syscall_filter, filter, sizeof(curproc->syscall_filter));
    curproc->trace_success = success_flag;
    curproc->trace_failure = failure_flag;
    curproc->next_trace_pid = curproc->pid + 1; // Target the next process spawned
    curproc->trace_flags_active = 1;           // Mark flags as active

    return 0;
}
```

```
// Reset the tracing flags after the targeted process completes
void reset_trace_flags(struct proc *curproc) {
    if (curproc->trace_flags_active && curproc->pid == curproc->next_trace_pid) {
        // Reset tracing flags
        curproc->syscall_filter[0] = '\0';
        curproc->trace_success = 0;
        curproc->trace_failure = 0;
        curproc->next_trace_pid = 0;
        curproc->trace_flags_active = 0;
    }
}
```

OUTPUT:



In this output, the `strace -e write` command filters system calls to only log `write` system calls for the first `echo hello`. It prints all `write` calls invoked by the process. For the next `echo hello`, all system calls (not just `write`) within the process are logged, showing the behavior without the filter. **This highlights the filtering mechanism of the `-e` option.**

4.3.2: Option: -s

`-s` (Log Successful System Calls): The `-s` option logs only system calls that succeed. This reduces unnecessary details and makes it easier to see what operations completed successfully in the program.

```
$ dsf
TRACE: pid = 5 | command_name = sh | syscall = sbrk | return value = 16384
exec: fail
TRACE: pid = 5 | command_name = sh | syscall = exec | return value = -1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 5 | command_name = sh | syscall = write | return value = 1
$ strace -s
TRACE: pid = 6 | command_name = sh | syscall = sbrk | return value = 16384
TRACE: pid = 6 | command_name = strace | syscall = exec | return value = 0
TRACE: pid = 6 | command_name = strace | syscall = set_trace_flags | return value = 0
$ dsf
TRACE: pid = 7 | command_name = sh | syscall = sbrk | return value = 16384
exec: fail
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 7 | command_name = sh | syscall = write | return value = 1
$
```

This output shows how we can filter and display only successful system calls. Initially, all system calls, including failed ones with return values of `-1`, are logged. Once the `strace -s` option is enabled, only successful system calls with positive return values are displayed. The flags are also reset after each command, ensuring the filter is applied only once per use, keeping the output focused and relevant.

4.3.3: Option: -f [5 points]

`-f` (Log Failed System Calls): The `-f` option shows only the system calls that failed. This is helpful for identifying errors or issues in the program's execution.

```
init: starting sh
$ strace on
TRACE: pid = 3 | command_name = strace | syscall = strace | return value = 0
$ strace -f
TRACE: pid = 4 | command_name = sh | syscall = sbrk | return value = 16384
TRACE: pid = 4 | command_name = strace | syscall = exec | return value = 0
TRACE: pid = 4 | command_name = strace | syscall = set_trace_flags | return value = 0
$ asdf
exec: fail
TRACE: pid = 5 | command_name = sh | syscall = exec | return value = -1
$ asdf
TRACE: pid = 6 | command_name = sh | syscall = sbrk | return value = 16384
exec: fail
TRACE: pid = 6 | command_name = sh | syscall = exec | return value = -1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 6 | command_name = sh | syscall = write | return value = 1
$
```

```
$ strace -f
TRACE: pid = 11 | command_name = sh | syscall = sbrk | return value = 16384
TRACE: pid = 11 | command_name = strace | syscall = exec | return value = 0
TRACE: pid = 11 | command_name = strace | syscall = set_trace_flags | return value = 0
$ sdaf
exec: fail
TRACE: pid = 12 | command_name = sh | syscall = exec | return value = -1
$
```

This output showcases the use of the `-f` option, which focuses on logging only failed system calls. For the first invalid command, `asdf`, it displays system calls with a return value of `-1`, indicating failure. When another invalid command is executed, it similarly logs only the failed system calls for that specific instance. The `-f` flag works as a one-time filter, ensuring clean and targeted output for each command.

4.3.4: Extra credits: Combine options [5 points]

```
$ strace -f -e exec
TRACE: pid = 6 | command_name = sh | syscall = sbrk | return value = 16384
TRACE: pid = 6 | command_name = strace | syscall = exec | return value = 0
TRACE: pid = 6 | command_name = strace | syscall = set_trace_flags | return value = 0
$ echh
exec: fail
TRACE: pid = 7 | command_name = sh | syscall = exec | return value = -1
$ 
```

This output highlights how the `-e` option works in filtering specific system calls. In the first example, `strace -f -e exec` is used, and since the command `echh` is invalid, it results in a failed `exec` system call, which is logged because it matches the `-e exec` filter. This shows how the option focuses on logging only the specified system call, even when it fails.

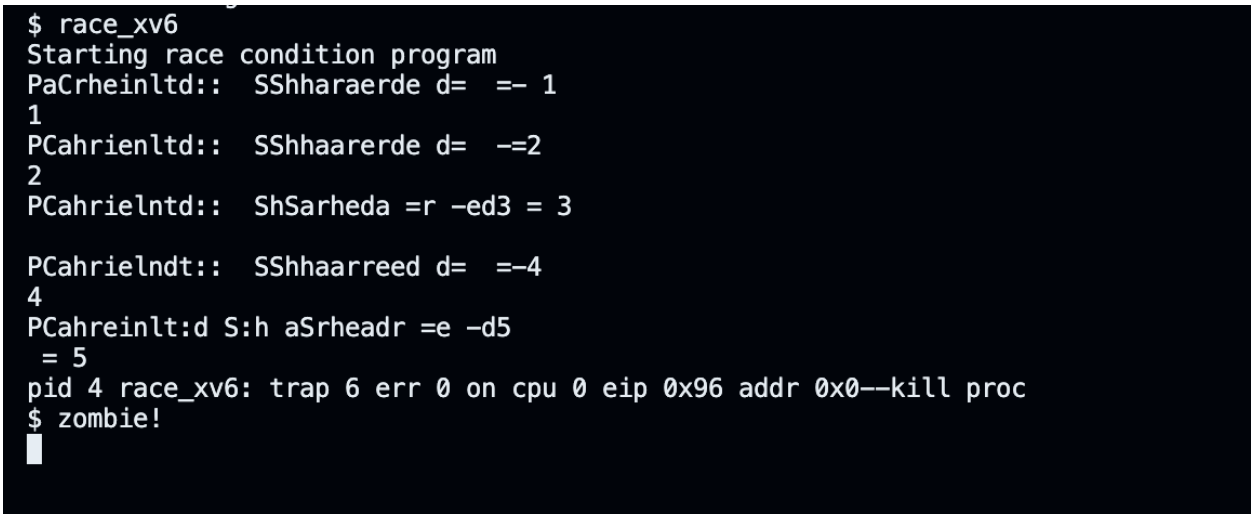
```
$ strace -s -e write
TRACE: pid = 21 | command_name = sh | syscall = sbrk | return value = 16384
TRACE: pid = 21 | command_name = strace | syscall = exec | return value = 0
TRACE: pid = 21 | command_name = strace | syscall = set_trace_flags | return value = 0
$ fsdaf
TRACE: pid = 22 | command_name = sh | syscall = sbrk | return value = 16384
exec: fail
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
TRACE: pid = 22 | command_name = sh | syscall = write | return value = 1
$ 
```

Here, `strace -s -e write` tracks only successful `write` system calls. For the command `fsdf`, the output shows all the `write` calls related to this command, while ignoring any unrelated or failed calls. This demonstrates how the `-e` option allows us to focus on specific system call outcomes, making the logs precise and relevant.

4.5 Application of Strace

The program shows race conditions ,memory leaks and an unexpected crash. The parent and child process access a shared variable without synchronization, causing unpredictable interleaving of outputs. The child process intentionally **crashes** by dereferencing a null pointer, while memory is allocated but never freed, simulating a leak. On xv6, the program crashes with a "trap 6" error and shows overlapping `write` syscalls using `strace`, revealing the race condition, but it lacks detailed crash or memory leak analysis. On Linux, the program runs without crashing due to how null pointer dereferencing is handled, and `strace` provides detailed logs for every syscall, including memory operations like `mmap` and `brk`, and clear crash diagnostics with `SIGSEGV`**. While xv6 is helpful for basic syscall tracing and identifying race conditions, Linux offers a more complete view, making it much better for debugging crashes and memory-related issues**.

Output of race_xv6 without strace on:



Output of race_xv6 with strace on

```
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = sleep | return value = 0
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = sleep | return value = 0
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 7 | command_name = race_xv6 | syscall = write | return value = 1
TRACE: pid = 8 | command_name = race_xv6 | syscall = write | return value = 1
unexpected trap 14 from cpu 0 eip 80104b28 (cr2=0x1010129)
lapicid 0: panic: trap
 80105fbf 80105cae 80104c02 80105f61 80105cae 0 0 0 0 0
```

output for race_linux without strace

```
@aru456 → /workspaces/xv6-public (aru-strace) $ ./race_linux
Starting race condition program
Parent: Shared = -1
Child: Shared = 1
Parent: Shared = -2
Child: Shared = 2
Parent: Shared = -3
Child: Shared = 3
Parent: Shared = -4
Child: Shared = 4
Parent: Shared = -5
Child: Shared = 5
@aru456 → /workspaces/xv6-public (aru-strace) $
```

output for strace ./race_linux

```
@aru456 → /workspaces/xv6-public (aru-strace) $ strace ./race_linux
execve("./race_linux", ["./race_linux"], 0x7ffdd9120670 /* 101 vars */) = 0
brk(NULL)                               = 0x5ba1d932d000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc3134e350) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=44733, ...}) = 0
mmap(NULL, 44733, PROT_READ, MAP_PRIVATE, 3, 0) = 0x72003ad9e000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300A\2\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\7\2C\n\357_\243\335\2449\206V>\237\374\304"..., 68, 880) = 68
fstat(3, {st_mode=S_IFREG|0755, st_size=2029592, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x72003ad9c000
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\7\2C\n\357_\243\335\2449\206V>\237\374\304"..., 68, 880) = 68
mmap(NULL, 2037344, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x72003abaa000
mmap(0x72003abcc000, 1540096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) = 0x72003abcc000
mmap(0x72003ad44000, 319488, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19a000) = 0x72003ad44000
mmap(0x72003ad92000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x72003ad92000
mmap(0x72003ad98000, 13920, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x72003ad98000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x72003ad9d540) = 0
mprotect(0x72003ad92000, 16384, PROT_READ) = 0
mprotect(0x5ba1d734a000, 4096, PROT_READ) = 0
mprotect(0x72003add6000, 4096, PROT_READ) = 0
munmap(0x72003ad9e000, 44733)           = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
brk(NULL)                               = 0x5ba1d932d000
brk(0x5ba1d934e000)                     = 0x5ba1d934e000
write(1, "Starting race condition program\n", 32Starting race condition program
) = 32
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x72003ad9d810) = 257227
write(1, "Parent: Shared = -1\n", 20Parent: Shared = -1
)   = 20
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, Child: Shared = 1
0x7ffc3134e2c0) = 0
write(1, "Parent: Shared = -2\n", 20Parent: Shared = -2
)   = 20
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, Child: Shared = 2
0x7ffc3134e2c0) = 0
write(1, "Parent: Shared = -3\n", 20Parent: Shared = -3
)   = 20
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, Child: Shared = 3
0x7ffc3134e2c0) = 0
write(1, "Parent: Shared = -4\n", 20Parent: Shared = -4
)   = 20
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, Child: Shared = 4
0x7ffc3134e2c0) = 0
write(1, "Parent: Shared = -5\n", 20Parent: Shared = -5
)   = 20
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, Child: Shared = 5
0x7ffc3134e2c0) = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Comparison:

xv6 strace provides basic syscall tracing, showing names, return values, and concurrency issues, but its crash diagnostics are limited to traps (trap 6, trap 14). Linux strace offers richer logs with syscall arguments, memory operations (mmap, brk), and detailed crash diagnostics (SIGSEGV). While xv6 is suitable for understanding basic syscall behaviors, Linux is better for identifying the root cause of crashes and memory-related issues.