Operating Systems Lab
Assignment 1
Group 4

Drive Link to Patch File:
https://drive.google.com/drive/folders/1lK-xRbZGjG1nzhi5WVVqiGUa_T3QcXZi?usp=drive_link

Group Members:

- Tarun Raj 220101104
- Ayush Savar 220101022
- Tanush Reddy Kolagatla 220101101
- Udbhav Gupta 220101106

Task 1.1

Objective

The objective of this assignment was to implement a user-level sleep program for the xv6 operating system. The program is designed to pause execution for a user-specified number of ticks. In xv6, a tick is a time unit defined by the kernel, representing the time between two interrupts from the timer chip.

Implementation Details

1. **Source Code Overview (`sleep.c`):**
    - The sleep.c file, located in the user/ directory, contains the implementation of the sleep command.
    - The program takes an argument specifying the number of ticks to sleep.
    - If no argument is provided, the program outputs an error message and usage instructions, then exits.
    - If a valid number of ticks is provided, the program converts the argument to an integer and calls the sleep() function to pause execution for the specified number of ticks.
    - After the sleep duration, the program exits gracefully.

Code:

```c
#include "types.h"
#include "user.h"
#include "stat.h"
int main(int argc, char *argv[])
{
    int ticks; // time to sleep

    if (argc <= 1)
    {
        printf(2, "Error: Missing argument.\nUsage: sleep <ticks>\n");
        printf(2, "Please specify the number of ticks to sleep as an argument.\n");
        exit();
    }
    ticks = atoi(argv[1]);

    sleep(ticks);

    exit();
}
```

2. Compiling the Sleep Program:

- The Makefile in xv6 was modified to include the new sleep program.
- The UPROGS variable was updated to compile and include the sleep program along with other user programs in the file system image.

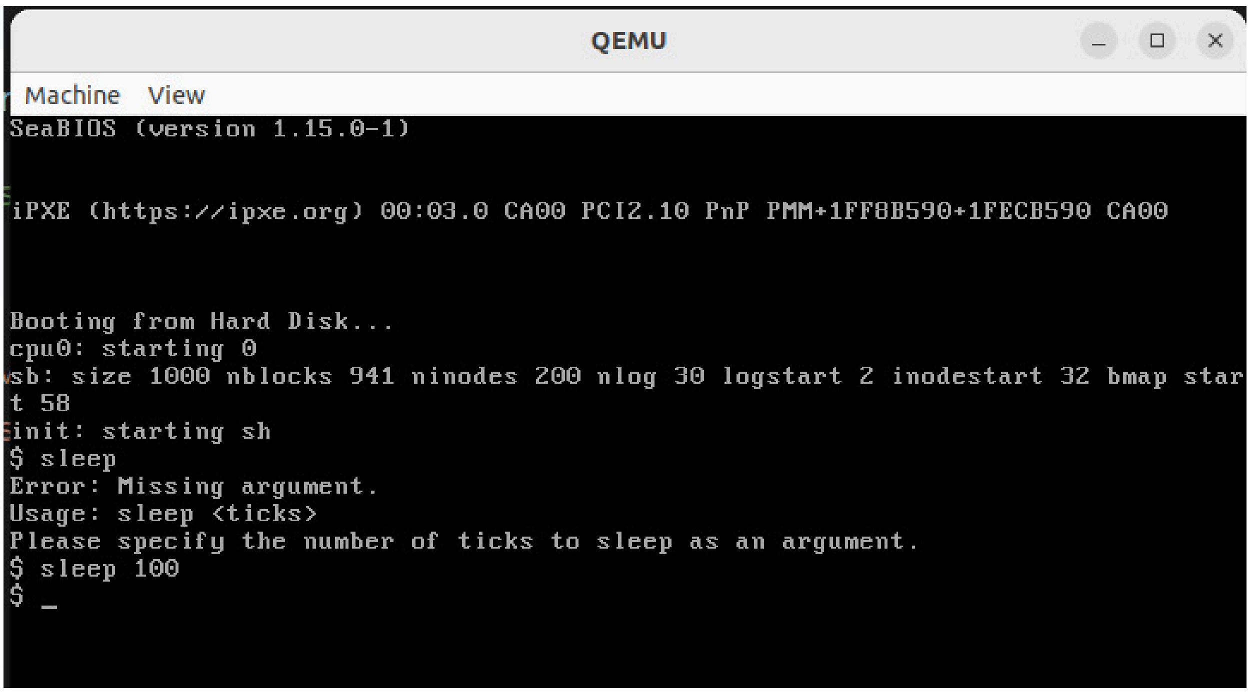
Makefile Update:

```makefile
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_bananacheck\
	_sleep\
	_animation\
	_wait2test\

fs.img: mkfs README $(UPROGS)
	./mkfs fs.img README $(UPROGS)

-include *.d
```

Testing the Sleep Program:

- The program was tested by running it in the xv6 QEMU environment.
- The first test case involved running the program without any arguments. The expected behavior is an error message indicating the missing argument, which was successfully observed in the QEMU terminal.
- The second test case provided a valid number of ticks (e.g., `100`) to sleep. The program correctly paused for the specified duration before returning to the shell prompt.

QEMU Terminal Output:



Conclusion

The sleep program was successfully implemented and integrated into the xv6 operating system. The program behaves as expected, handling both error cases (missing argument) and normal operation (sleeping for the specified number of ticks). The modified Makefile ensures that the program is included in the xv6 file system image, making it available for use in the xv6 shell environment.

Task 1.2

Objective

The task was to create a user-level animation program for the xv6 operating system. The program should display an animated sequence in the terminal, which cycles through different frames with a pause between each. The animation continues to loop indefinitely. When the OS runs, the program's binary should be included in `fs.img`, and it should be listed when someone runs `ls` at the xv6 shell's command prompt.

Implementation Details

1. **Source Code Overview (`animation.c`):**
   - The `animation.c` file, located in the `user/` directory, contains the implementation of the animation command.
   - The program uses two frames of ASCII art that alternate to create a simple animation.
   - A helper function `clear_screen()` is used to clear the screen between 2 frames, creating the effect of a moving image.
   - The program continuously loops through the frames with a delay between each using the `sleep()` function to pause execution for a specified number of ticks.
   - The `exit()` function is called to terminate the program if necessary (though in this case, it runs indefinitely).

Code:

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

void clear_screen()
{
   // This is a simple way to clear the screen by printing many newlines
   printf(1, "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
}

int main(int argc, char *argv[])
{
   char frame1[] = "\n\
      ROFL:ROFL:ROFL:ROFL\n\
       _^_ _\n\
   L    __/      [] \\\n\
   LOL===__           \\\n\
   L      \\_ _ _]\n\
            I   I\n\
         ----------/\n\n";

   char frame2[] = "\n\
            :          \n\
       _^_ _\n\
   .    __/      [] \\\n\
   L..===__           \\\n\
   .      \\_ _ _]\n\
            I   I\n\
         ----------/\n\n";

   int cur_frame = 1;
   while (1)
   {
      clear_screen();
      if (cur_frame == 1)
      {
         printf(1, frame1);
         cur_frame++;
         sleep(15);
      }
      else if (cur_frame == 2)
      {
         printf(1, frame2);
         cur_frame = 1;
         sleep(15);
      }
   }

   exit();
}
```

2. Compiling the Animation Program:

- Similar to the sleep utility, the Makefile was modified to include the animation program.
- The UPROGS variable was updated to compile and include the animation program along with other user programs in the file system image.

Makefile Update:

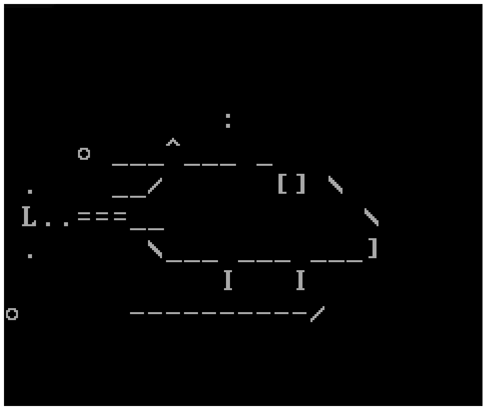
```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_bananacheck\
	_sleep\
	_animation\
	_wait2test\

fs.img: mkfs README $(UPROGS)
	./mkfs fs.img README $(UPROGS)

-include *.d
```

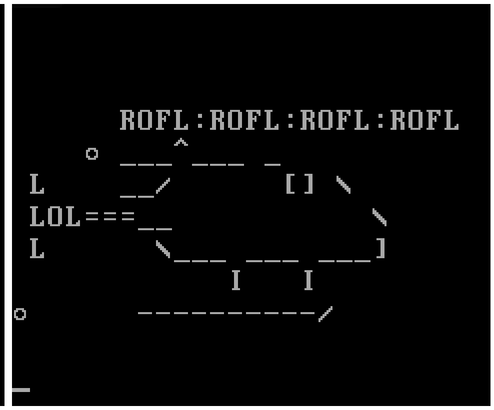
3. Testing the Animation Program:

- The program was tested by running it in the xv6 QEMU environment.
- Upon execution, the program displayed an animated sequence by alternating between two ASCII art frames with a pause of 15 ticks between each frame.
- The animation continued to loop indefinitely, creating a simple visual effect.

4. Screenshots:

- The following screenshots were captured to demonstrate the animation in the xv6 QEMU terminal.





Conclusion

The animation program was successfully implemented and integrated into the xv6 operating system. The program performs as expected, continuously looping through a sequence of ASCII art frames with a pause between each. The modified Makefile ensures that the program is included in the xv6 file system image and is available for use in the xv6 shell environment.

Task 1.3:

In this task, we implemented an infrastructure to collect and report statistics on process states within an operating system. This infrastructure is critical for evaluating the performance of various scheduling policies that we may implement in future tasks. The focus was on extending the proc structure to track time spent in different process states (SLEEPING, READY, RUNNING) and implementing a new system call wait2 to retrieve this information.

Changes Made

The changes involved modifications to several source files in the operating system codebase:

1. **Extending the proc Structure (proc.h and proc.c)**
   - **Purpose**: The proc structure was extended to include four new fields: ctime, stime, retime, and rutime. These fields represent the creation time, sleep time, ready time, and run time of each process, respectively.
   - **Details**:
     - **ctime**: Set when a process is created.
     - **stime**: Incremented when a process is in the SLEEPING state (waiting for I/O).
     - **retime**: Incremented when a process is in the READY (RUNNABLE) state but not running.
     - **rutime**: Incremented when a process is actively running.

```
// Per-process state
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 stime;                   // sleep time
  int rutime;                  // total run time
  int ctime;                   // creation time
  int retime;                  // ready time
};
```

Handling Clock Ticks (trap.c)

- **Purpose**: To update the process state times on each clock tick, depending on the current state of the process.
- **Details**:

    - The trap function was modified to handle the T_IRQ0 + IRQ_TIMER interrupt.
    - The function checks the state of the current process (RUNNING, SLEEPING, or RUNNABLE) and increments the corresponding time field (rutime, stime, or retime).

Code Excerpt from trap.c:

```c
void trap(struct trapframe *tf)
{
  if (tf->trapno == T_SYSCALL)
  {
    if (myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if (myproc()->killed)
      exit();
    return;
  }

  switch (tf->trapno)
  {
  case T_IRQ0 + IRQ_TIMER:
    ticking();
    if (cpuid() == 0)
    {
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
```

Implementation of the `wait2` System Call (`sysproc.c` and `proc.c`)

- **Purpose**: To allow user programs to retrieve the accumulated statistics for terminated child processes.
- **Details**:
    - A new system call `wait2` was implemented, which takes pointers to `retime`, `rutime`, and `stime` as arguments.

- The `wait2` system call behaves similarly to the existing `wait` call but additionally returns the aggregated time spent in each state.
- The actual implementation of the `wait2` function in `proc.c` iterates over the process table, finds the child process that has terminated, collects the statistics, and then returns the PID of the terminated process.

Code Excerpt from `sysproc.c`:

```
int sys_wait2(void)
{
  int *retime, *rutime, *stime;
  if (argptr(0, (char **)&retime, sizeof(int)) < 0 ||
      argptr(1, (char **)&rutime, sizeof(int)) < 0 ||
      argptr(2, (char **)&stime, sizeof(int)) < 0)
    return -1;
  return wait2(retime, rutime, stime);
}
```

Code Excerpt from `proc.c`:

```c
int wait2(int *retime, int *rutime, int *stime)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for (;;)
  {
    // Scan through table looking for exited children.
    havekids = 0;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->parent != curproc)
        continue;
      havekids = 1;

      *retime = p->retime;
      *rutime = p->rutime;
      *stime = p->stime;
      if (p->state == ZOMBIE)
      {
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if (!havekids || curproc->killed)
    {
      release(&ptable.lock);
      return 1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock); // DOC: wait-sleep
  }
}
```

Adding wait2 to the System Call Interface (syscall.c and syscall.h)

- **Purpose:** To integrate the new wait2 system call into the system call interface.
- **Details:**
  - The system call number for wait2 was added to syscall.h.
  - The corresponding entry was added to syscall.c to link the system call number with the sys_wait2 function in sysproc.c.

Code Excerpt from syscall.h:

```
// System call numbers
#define SYS_fork 1
#define SYS_exit 2
#define SYS_wait 3
#define SYS_pipe 4
#define SYS_read 5
#define SYS_kill 6
#define SYS_exec 7
#define SYS_fstat 8
#define SYS_chdir 9
#define SYS_dup 10
#define SYS_getpid 11
#define SYS_sbrk 12
#define SYS_sleep 13
#define SYS_uptime 14
#define SYS_open 15
#define SYS_write 16
#define SYS_mknod 17
#define SYS_unlink 18
#define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21
#define SYS_banana 22
#define SYS_wait2 23
```

Code Excerpt from `syscall.c`:

```
extern int sys_wait2(void);
```

Testing the wait2 System Call (wait2test.c)

- **Purpose**: To verify the correct implementation of the wait2 system call.
- **Details**:
  - A test program was written to create a child process and simulate its running, ready, and sleeping states.

- The parent process calls wait2 after the child terminates and prints the returned statistics.
- The results were used to validate the accuracy of the ctime, stime, retime, and rutime fields.

Changes in `proc.c`:

The code snippet we have shared introduces a function named `ticking` that updates the process runtime statistics. Here's a breakdown:

```
void ticking()
{
  struct proc *process;
  acquire(&ptable.lock);

  process = ptable.proc;
  for (; process < &ptable.proc[NPROC]; process++)
  {
    if (process->state == RUNNING)
      process->rutime++;
    if (process->state == RUNNABLE)
      process->retime++;
    if (process->state == SLEEPING)
      process->stime++;
  }

  release(&ptable.lock);
}
```

`ticking()` Function:

- **Purpose:**
  1. To update the time spent by each process in different states (RUNNING, RUNNABLE, SLEEPING).
  2. This helps in tracking how much time each process spends in each state, which is crucial for performance evaluation.
- **Function Logic:**
  1. **Locking the Process Table:**

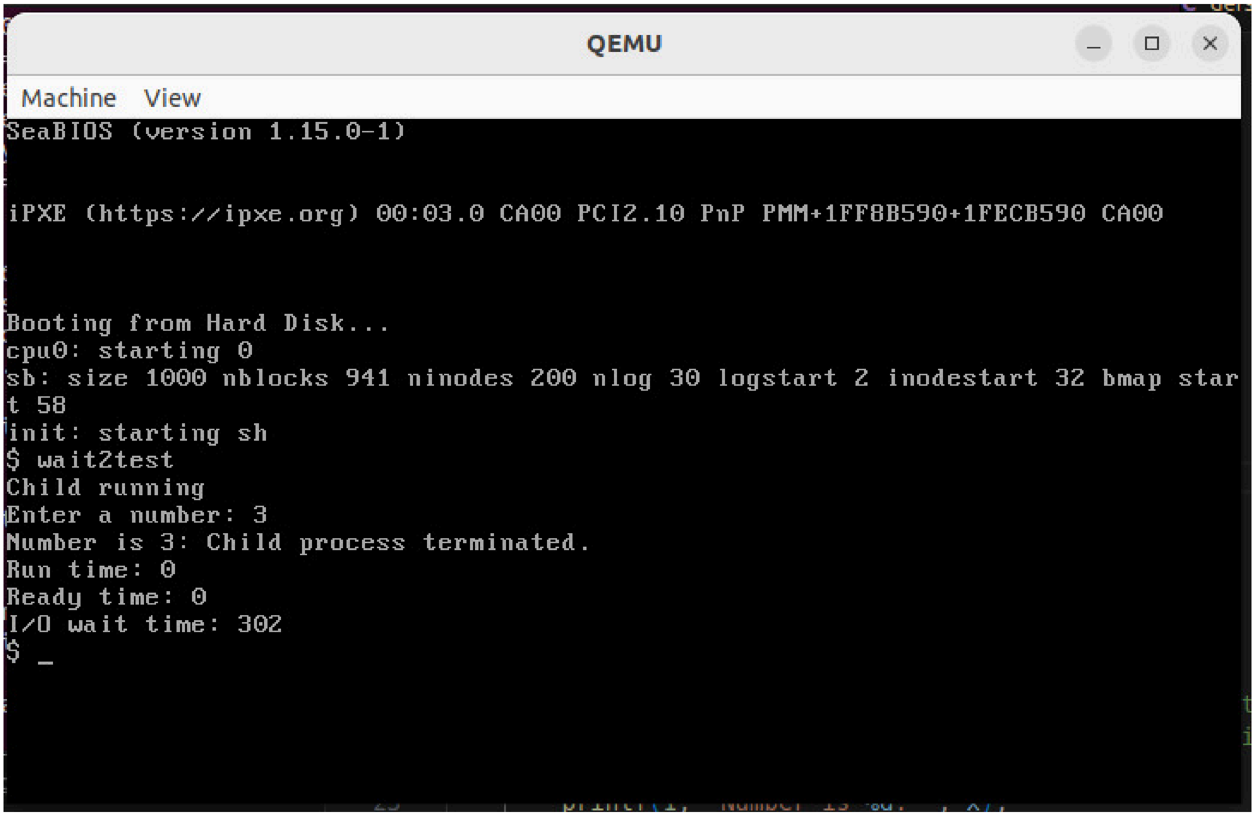
        - `acquire(&ptable.lock);` ensures that the process table (`ptable`) is not modified by other processes while `ticking()` is updating the process times.
2. **Loop through Processes:**
    - `process = ptable.proc;` initializes a pointer to the start of the process table.
    - The `for` loop iterates over all processes in the system (`NPROC` is the maximum number of processes).
3. **State-Based Updates:**
    - If a process is `RUNNING`, it increments its `rutime` (running time).
    - If a process is `RUNNABLE`, it increments its `retime` (ready time).
    - If a process is `SLEEPING`, it increments its `stime` (I/O wait time).
4. **Releasing the Lock:**
    - `release(&ptable.lock);` ensures that other processes can now access and modify the process table.

This `ticking` function should be called periodically, likely on every clock tick, to keep the process time statistics up to date.

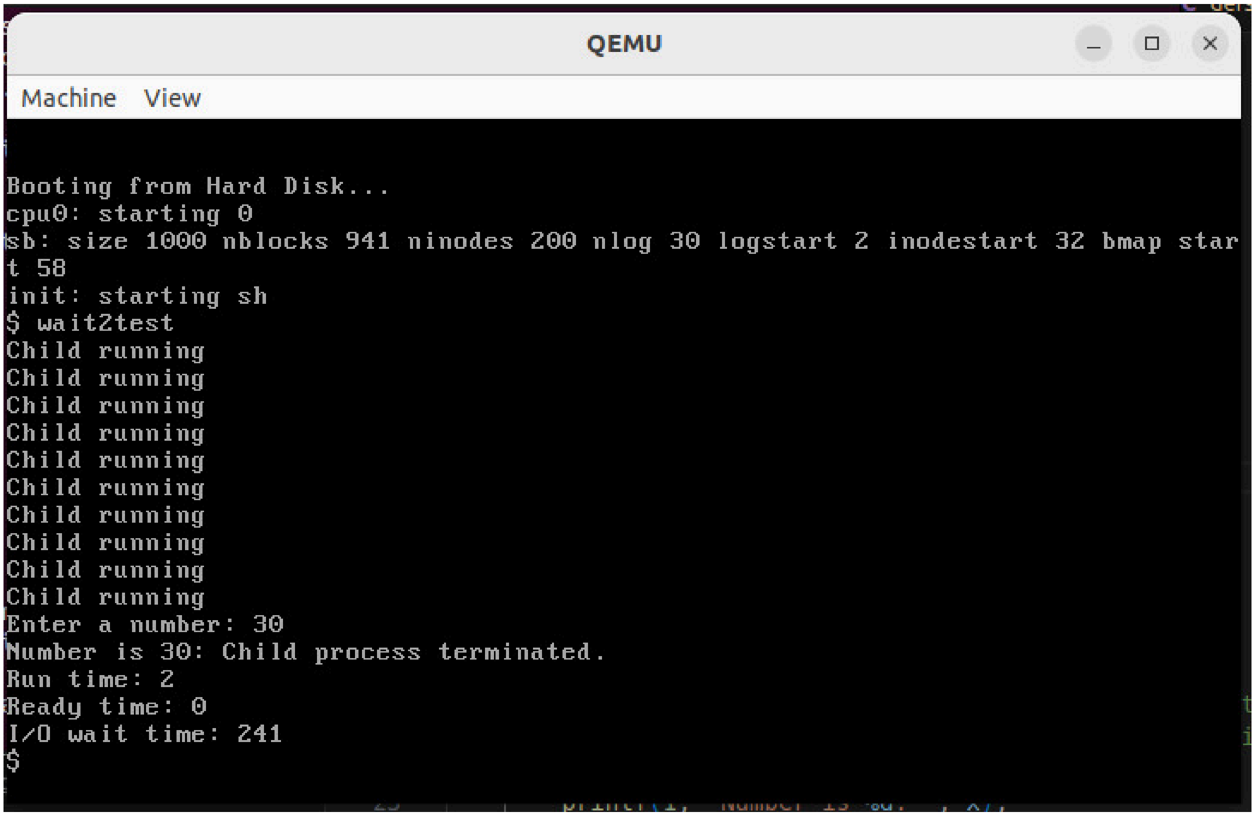
Changes in `defs.h`:

- **Declaration of `ticking(void)` Function:**
    - In `defs.h`, we needed to add the line `void ticking(void);` which is a forward declaration of the `ticking` function. This ensures that other parts of the kernel can invoke `ticking()` even before its implementation is encountered.

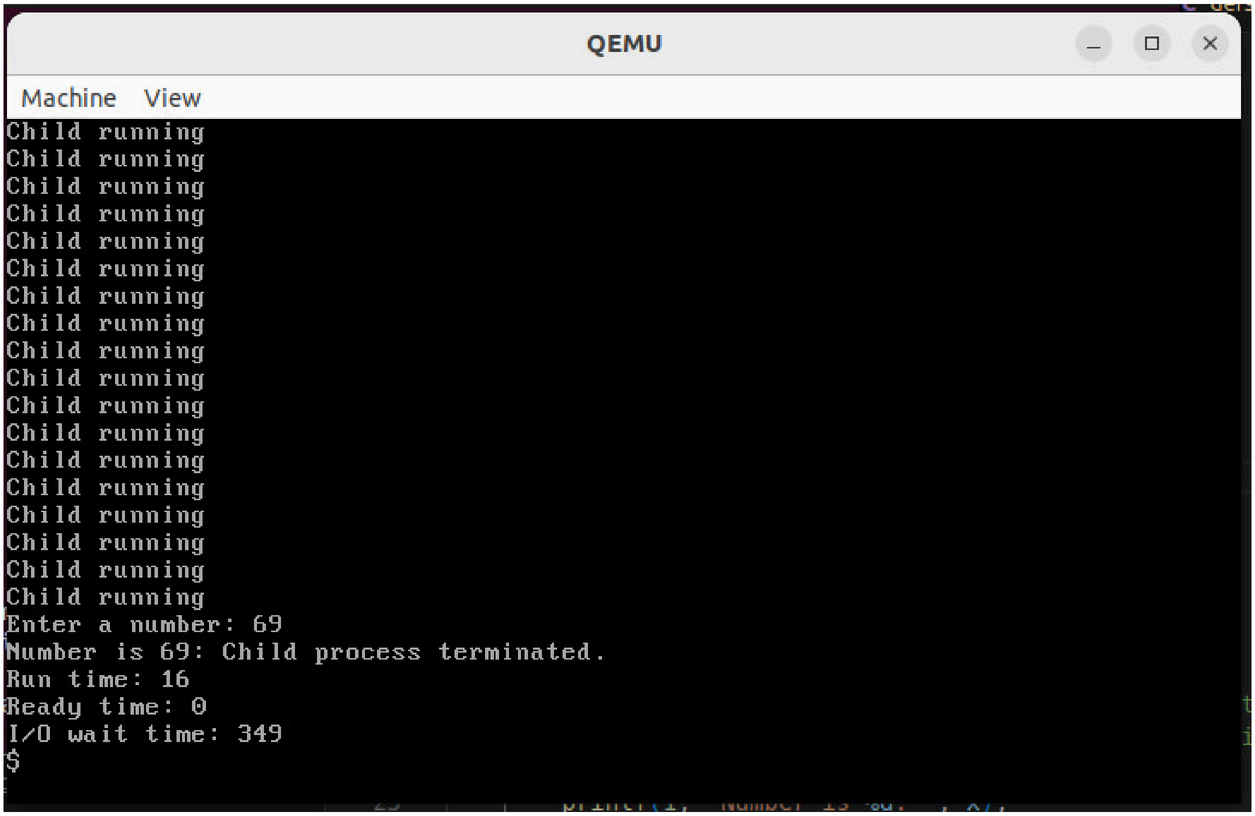
```
void ticking(void);
```



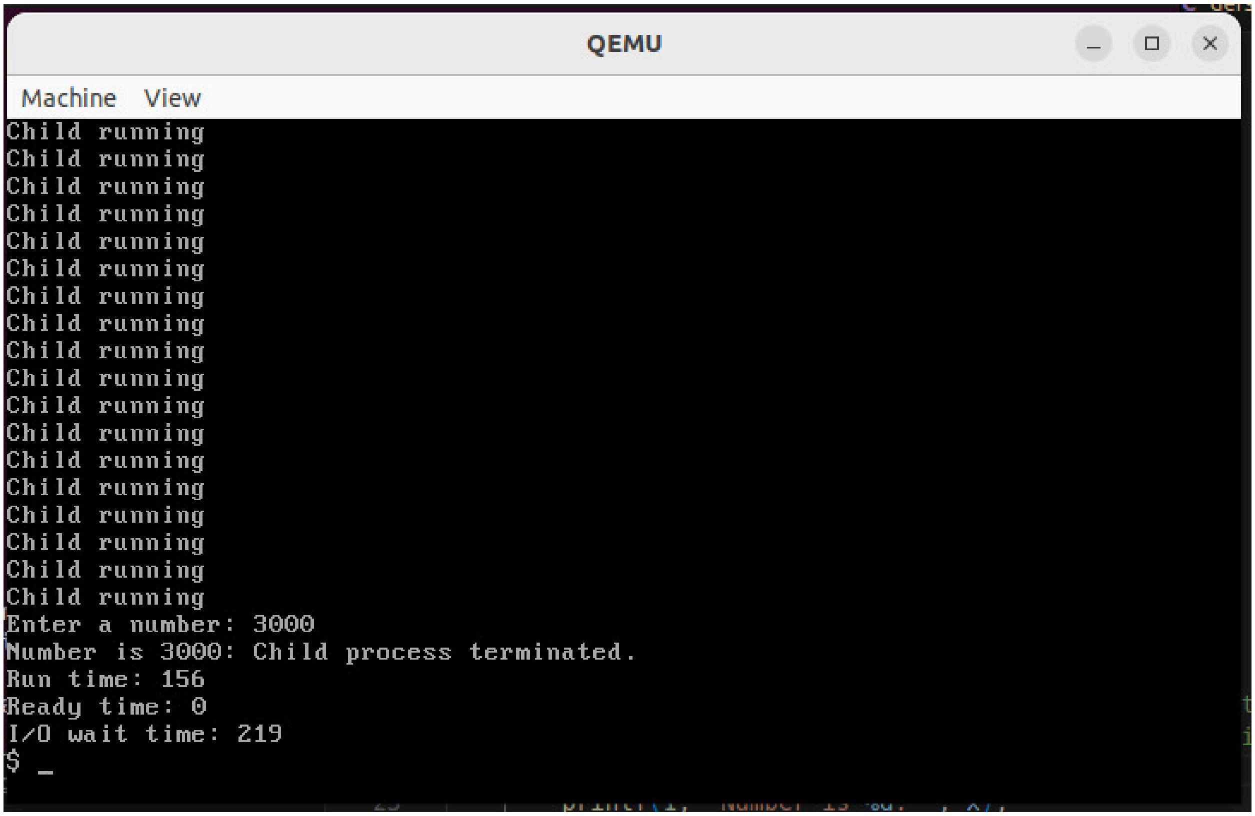
- **Entered Number:** 3
- **Run Time:** 0 clock ticks
- **I/O Wait Time:** 302 clock ticks
- **Description:** The child process terminates quickly with minimal CPU work, resulting in zero runtime. However, it experiences significant I/O wait time, likely due to blocking or waiting for resources.



- **Entered Number:** 30
- **Run Time:** 2 clock ticks
- **I/O Wait Time:** 241 clock ticks
- **Description:** The increased entered number leads to more CPU activity, slightly increasing the runtime. The I/O wait time decreases slightly, possibly due to a change in the process's blocking behavior.



- **Entered Number:** 69
- **Run Time:** 16 clock ticks
- **I/O Wait Time:** 349 clock ticks
- **Description:** A larger entered number significantly increases the process's workload, resulting in a substantial rise in runtime. The I/O wait time also increases, indicating more time spent in blocking or waiting states.



- **Entered Number:** 3000
- **Run Time:** 156 clock ticks
- **Ready Time:** 0 clock ticks
- **I/O Wait Time:** 219 clock ticks
- **Description:** With an extremely large entered number, the process consumes a substantial amount of CPU time, reflected in the high runtime. Despite this, the I/O wait time remains significant, while the ready time is zero, indicating no delays in transitioning from ready to running states.

Increase in Wait Time and Runtime:

- **Wait Time (`stime`):**
  - The entered number in your program likely affects the time a process spends in the `SLEEPING` state. For example, if a large number is entered, the process might perform more I/O operations or might be blocked waiting for a resource, increasing the I/O wait time.
- **Runtime (`rutime`):**

    - The amount of loops in your code directly correlates with the time the process spends actively executing instructions (i.e., `RUNNING`). More loops mean more CPU cycles consumed, which increases the runtime.

Summary:

- **`proc.c`:** The `ticking()` function updates process state times at each clock tick, helping in accurate tracking of process behavior.
- **`defs.h`:** The `ticking(void);` declaration allows the kernel to recognize and call this function from other files.
- **Time Increases:** Larger numbers increase the wait time due to more I/O or blocking operations, while more loops increase the runtime due to additional CPU work.