1. Tasks

1.1 sleep

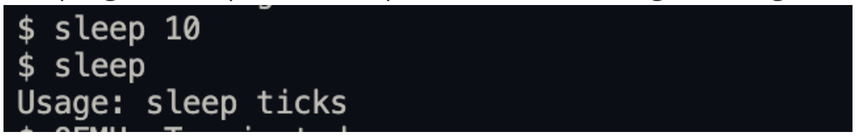
Implement a program to sleep for specified ticks: `sleep ticks`.

1.1.1 Procedure

Simply sleep via system call `sleep` after parsing the duration argument by `atoi`.

1.1.2 Result

The program will pause as required, and show usage message when duration is not specified.



```
$ sleep 10
$ sleep
Usage: sleep ticks
```

1.2 pingpong

Implement a program to transfer a byte of message between 2 processes through pipe.

1.2.1 Procedure

The main code is as follows (error handling is removed for simplicity):

```c
int main(int argc, char *argv[]) {
  int ppid, cpid;
  char buf[2], *info = "a";
  int p2c[2], c2p[2];
  pipe(p2c); pipe(c2p);

  ppid = getpid();

  if ((cpid = fork()) == 0) { // child
    cpid = getpid();
    close(p2c[1]);
    read(p2c[0], buf, 1);
    printf("%d: received ping from pid %d\n", cpid, ppid);
    close(c2p[0]);
    write(c2p[1], buf, 1);
    close(c2p[1]);
    exit(0);
  }
  // parent
  close(p2c[0]);
  write(p2c[1], info, 1);
  close(p2c[1]);
  wait(0);
  close(c2p[1]);
```

```
    read(c2p[0], buf, 1);
    printf("%d: received pong from pid %d\n", ppid, cpid);
    exit(0);
  }
```

The parent first sends the ping to child through pipe `p2c`, the child receives that and sends a pong back to parent through another pipe `c2p`, the parent reveices the pong then.

The following code uses only one pipe but also works well, which conflicts with question 2.3. No idea if this is an UB.

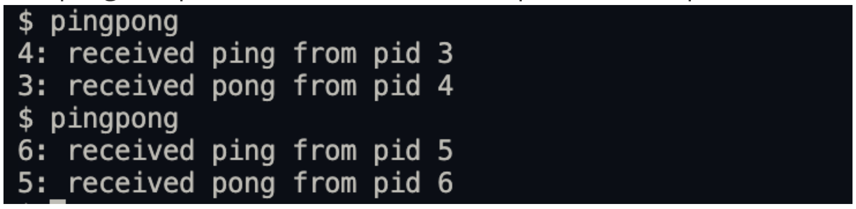
```c
int main(int argc, char *argv[]) {
  int ppid, cpid;
  char buf[2], *info = "a";
  int p[2]; pipe(p);

  ppid = getpid(); // parent pid

  if ((cpid = fork()) == 0) { // child
    cpid = getpid(); // child pid
    read(p[0], buf, 1); // 2. read ping from parent
    close(p[0]);
    printf("%d: received ping from pid %d\n", cpid, ppid);
    write(p[1], buf, 1); // 3. write pong to parent
    close(p[1]);
    exit(0);
  }
  // parent
  write(p[1], info, 1); // 1. write ping to child
  close(p[1]);
  wait(0); // wait for child to terminate
  read(p[0], buf, 1); // 4. read pong from child
  printf("%d: received pong from pid %d\n", ppid, cpid);
  close(p[0]);
  exit(0);
}
```

1.2.2 Result

The program prints the communication process with parent's and child's pid as required.



```
$ pingpong
4: received ping from pid 3
3: received pong from pid 4
$ pingpong
6: received ping from pid 5
5: received pong from pid 6
```

1.3 find

Implement a program to find target file or folder in some directory: `find dir file`.

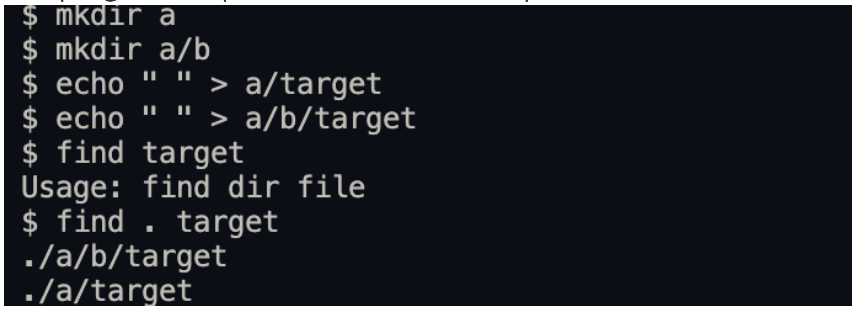
1.2.1 Procedure

The main logic is implemented via an auxiliary function `aux`, which recurs to perform a dfs search on the tree-like structure of a directory, described as follows:

```
void aux(char const *path, char const *target) {
  /* var declarations */
  fd = open(path, 0);
  /* copy path to buf */
  while (read(fd, &de, sizeof(de)) == sizeof(de)) {
    if (/* invalid entry */) continue;
    /* concat (buf + name) to build current file's path */
    /* check if is a match */
    if (/* current file is a directory */)
      aux(buf, target);
  }
  close(fd);
}
```

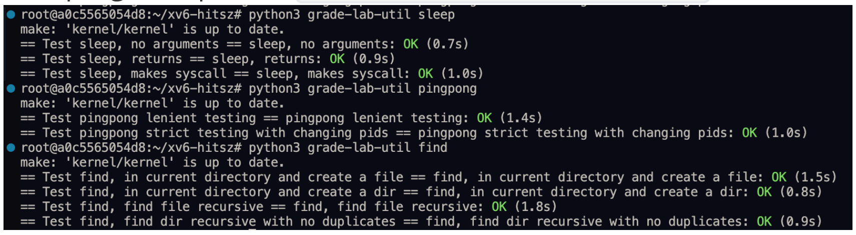
1.2.2 Result

The program outputs the matches as required.



```
$ mkdir a
$ mkdir a/b
$ echo " " > a/target
$ echo " " > a/b/target
$ find target
Usage: find dir file
$ find . target
./a/b/target
./a/target
```

All 3 programs passed `python3 grade-lab-util find` tests:



```
root@a0c5565054d8:~/xv6-hitsz# python3 grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (0.7s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
root@a0c5565054d8:~/xv6-hitsz# python3 grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong lenient testing == pingpong lenient testing: OK (1.4s)
== Test pingpong strict testing with changing pids == pingpong strict testing with changing pids: OK (1.0s)
root@a0c5565054d8:~/xv6-hitsz# python3 grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory and create a file == find, in current directory and create a file: OK (1.5s)
== Test find, in current directory and create a dir == find, in current directory and create a dir: OK (0.8s)
== Test find, find file recursive == find, find file recursive: OK (1.8s)
== Test find, find dir recursive with no duplicates == find, find dir recursive with no duplicates: OK (0.9s)
```

1.4 xv6 Booting

1.4.1 Steps

We know that during the booting session, xv6 will enter the `scheduler`, a function that never returns, from `main`. `scheduler` then schedules `initcode`, the initial process whose only job is to run `exec("/init")`. `init` is the ancestor of all processes.

This knowledge gives us some hint on how to print the name of `initcode` and `init`. After `make qemu-gdb` and `gdb` in another window, I started by setting a breakpoint at `scheduler`. When `continue` and hitting that breakpoint, I made a few `next` to see when would the process become `initcode`. It turned out when running `swtch(&c->context, &p->context);` the process is switched to `initcode`. So we can simplify this to setting a breakpoint at `swtch`.

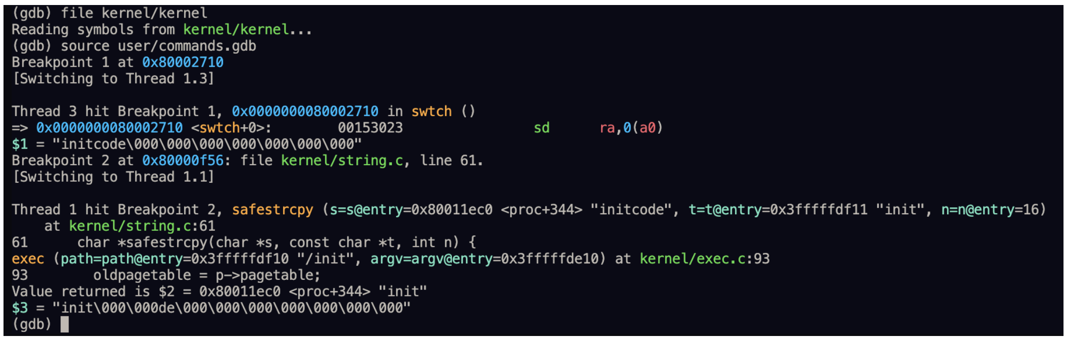
Next, we need to print the process `init`. Finding that `swtch` will be called several times (as it is in a loop), we need to remove that breakpoint now. Then I set a breakpoint at `exec`, since `initcode` will call that to let `init` replace it. After `continue`, the breakpoint is hit as expected. We can jump to its returning with a single `finish` command. Afterwards, `p cpus[$tp]->proc->name` will print the name of `init` properly.

The content of `commands.gdb` is as follows:

```
break swtch
continue
p cpus[$tp]->proc->name
delete 1
break exec
continue
finish
p cpus[$tp]->proc->name
```

(Sometimes an additional command `file kernel/kernel` is needed to specify the executable.)

1.4.2 Result



```
(gdb) file kernel/kernel
Reading symbols from kernel/kernel...
(gdb) source user/commands.gdb
Breakpoint 1 at 0x80002710
[Switching to Thread 1.3]

Thread 3 hit Breakpoint 1, 0x0000000080002710 in swtch ()
=> 0x0000000080002710 <swtch+0>:        00153023                sd      ra,0(a0)
$1 = "initcode\000\000\000\000\000\000\000"
Breakpoint 2 at 0x80000f56: file kernel/string.c, line 61.
[Switching to Thread 1.1]

Thread 1 hit Breakpoint 2, safestrcpy (s=s@entry=0x80011ec0 <proc+344> "initcode", t=t@entry=0x3fffffdf11 "init", n=n@entry=16)
    at kernel/string.c:61
61      char *safestrcpy(char *s, const char *t, int n) {
exec (path=path@entry=0x3fffffdf10 "/init", argv=argv@entry=0x3fffffde10) at kernel/exec.c:93
93          oldpagetable = p->pagetable;
Value returned is $2 = 0x80011ec0 <proc+344> "init"
$3 = "init\000\000de\000\000\000\000\000\000\000"
(gdb) 
```

2. Questions

2.1 How is the pipe created in the pingpong program?

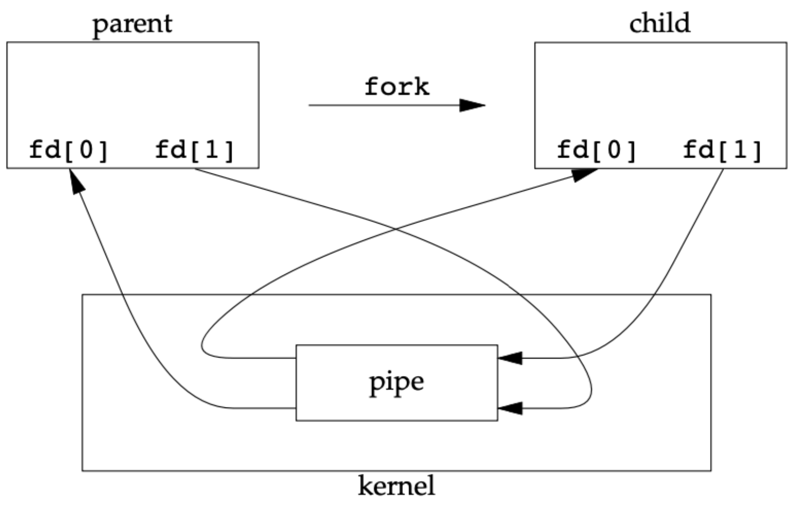
Declare a length-2 int array `int p[2];` and make it a pipe by system call `pipe(p);`. Then `p[0]` and `p[1]` may be used as if they are file descriptors (the former is the read end and the latter is the write end).

2.2 How is the data transferred between parent & child process via pipe?

Since the declaration of pipe `p` is before `fork()`, it is shared by the parent process and the child process. In the communication from parent to child, the parent writes data into the write end (then close it to signal finishing), and the child reads from the read end. (In theory one should close the pipe's read end before writing into it and vise versa, but in my case this works well). The IO of a pipe is nothing different from file IO on API level, as everything is a file in UNIX. The communication from child to parent is similar.

2.3 Why should the unused end of a pipe be closed in ahead?

The blocking mechanism of pipe says the pipe will block reading until its write end is closed. Suppose a pipe is used to read data in a process, if the write end is not closed, the process will never finish reading because there will always exist a opening write end, as suggested in the graph. Similarly, if a pipe is used to write data into in a process, if the read end remains open, it may accidentally reads the data it has written into so that the other process waiting to read data from the same pipe may not receive that data properly.



2.4 What would happen if the child process did not close the write end of pipe in the following program?

```
int p[2];
char *argv[2];
argv[0] = "wc";
argv[1] = 0;
pipe(p);

if (fork() == 0) {
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    exec("/bin/wc", argv);
} else {
    close(p[0]);
    write(p[1], "hello world\n", 12);
    close(p[1]);
}
```

If the child did not close the write end, i.e., `close(p[1]);`, the `wc` process would be hang: it would wait indefinitely for input because the pipe's write end remains open. Nothing would be output then.