# CS - 344 (OS LAB)

# Assignment - 1

# Group 1

**Members :**

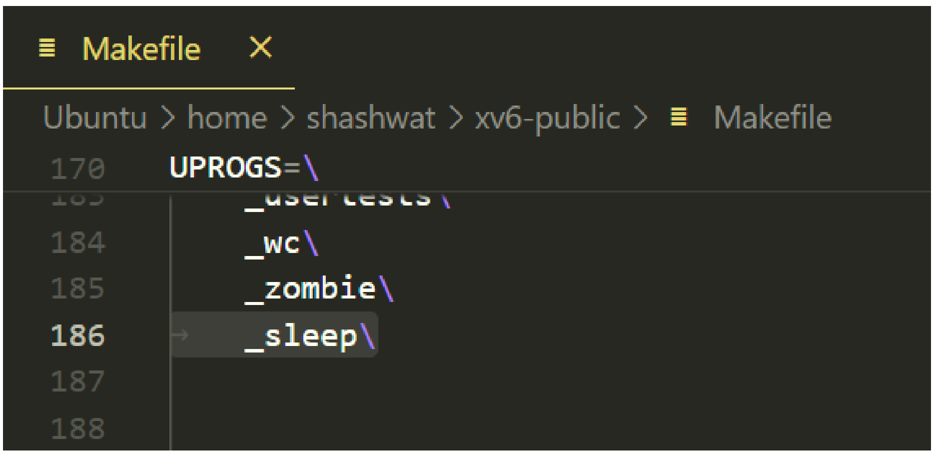
1. Parth Agarwal, CSE, 220101074
2. Shashwat Shankar, CSE, 220101092
3. Shobhit Gupta, CSE, 220101093
4. Shubhranshu Pandey, CSE, 220101094

**Drive Link:**

Group 1 OS Lab 1

## Task-1.1

1. Firstly, we create a new file called sleep.c in the user/ directory of the xv6 repository.
2. This new process is added to the process list (UPROGS) in Makefile.



The created sleep program :

sleep.c

Ubuntu > home > shashwat > xv6-public > sleep.c

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

int main(int argc, char *argv[]) {
    if (argc != 2) {  // Check if the user passed exactly one argument
        printf(1, "Incorrect number of arguments! Usage: sleep <ticks>\n");
        exit();
    }

    int ticks = atoi(argv[1]);  // Convert the argument from string to integer

    if(ticks == 0){ // atoi returns 0 for negative or invalid number
        printf(1, "sleep: invalid number of ticks\n");
        exit();
    }

    sleep(ticks);  // Call the sleep system call with the number of ticks

    exit();
}
```

3. The necessary header files are included first.

4. The program checks that the correct number of arguments are passed. If an incorrect number of arguments are passed, then we display appropriate error messages to the user and exit().

5. If the correct number of arguments are passed, atoi function is used to convert a string representing the number of ticks to the corresponding integer.

6. The sleep system call is then used to halt the execution for the desired number of ticks.

7. Finally, the program exits using exit() system call.

## Testing the Program

1. The program can be called by giving the command on the shell to run sleep. The number of ticks to sleep for is passed as the only argument.

2. The prompt doesn't appear immediately on the shell, instead showing up after the desired number of ticks. This shows that the created program is working correctly.

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ sleep 3000
_
```

Below, the behavior of the 'sleep' is shown when incorrect arguments are passed.

- If the number of arguments passed is not 1, then an error message is displayed to the user, along with telling him the correct usage.

```
$ sleep 50 50
Incorrect number of arguments! Usage: sleep <ticks>
$ _
```

- As the number of ticks to sleep for should be a non-negative integers, an appropriate message is displayed to the user in case of invalid argument.

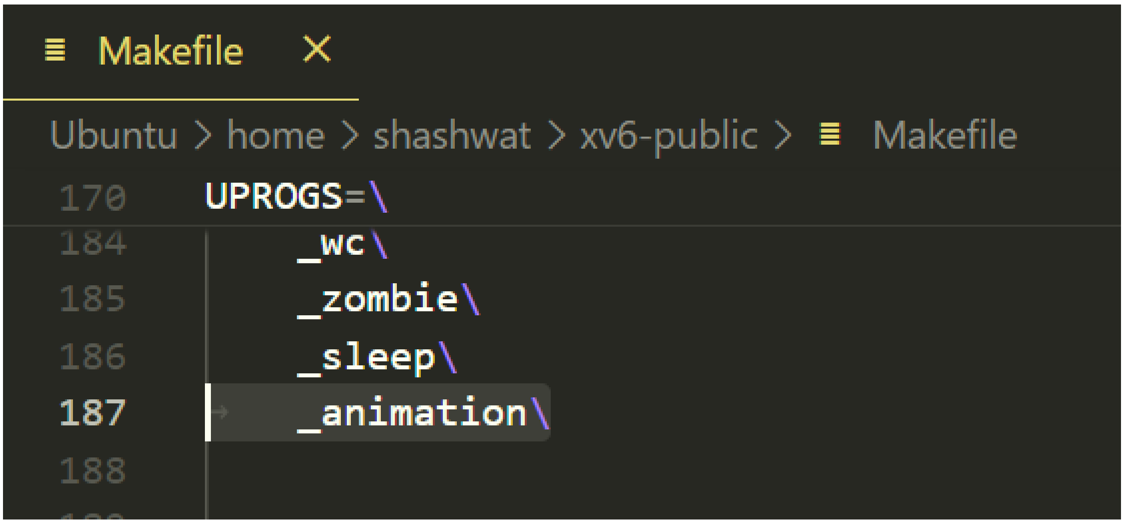
```
$ sleep -500
sleep: invalid number of ticks
$ _
```

```
$ sleep abcd
sleep: invalid number of ticks
$ _
```

# Task-1.2

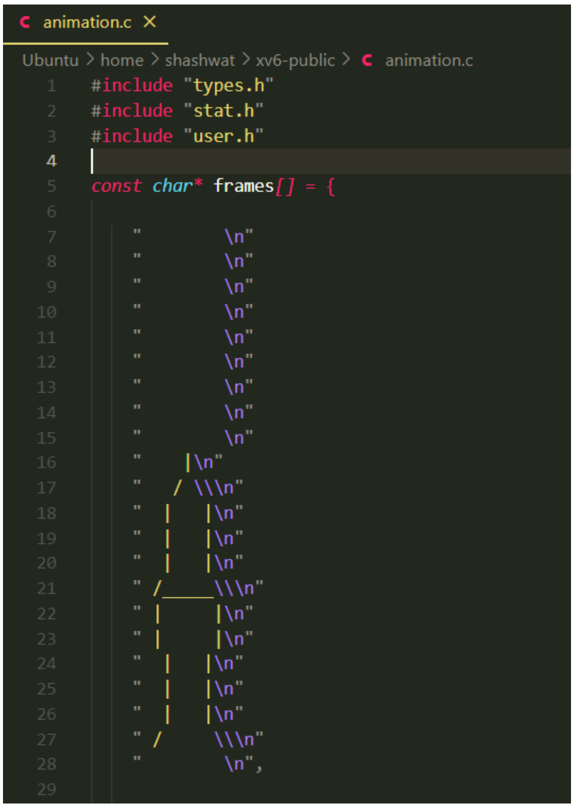
1. Firstly, we create a new file called animation.c in the user/ directory of the xv6 repository.

2. This new process is added to  the process list (UPROGS) in Makefile.



```
Makefile
Ubuntu > home > shashwat > xv6-public > Makefile
UPROGS=\
	_wc\
	_zombie\
	_sleep\
	_animation\
```

The created animation program of a launching rocket:



```c
animation.c
Ubuntu > home > shashwat > xv6-public > animation.c
#include "types.h"
#include "stat.h"
#include "user.h"

const char* frames[] = {

    "        \n"
    "        \n"
    "        \n"
    "        \n"
    "        \n"
    "        \n"
    "        \n"
    "        \n"
    "        \n"
    "    |\n"
    "   / \\\n"
    "  |   |\n"
    "  |   |\n"
    "  |   |\n"
    " /_____\\\n"
    " |     |\n"
    " |     |\n"
    "  |   |\n"
    "  |   |\n"
    "  |   |\n"
    " /     \\\n"
    "        \n",

```

```c
};

void clear_screen() {
    clear();
}

void draw_frame(int frame_index) {
    clear_screen();
    printf(1,"Frame %d\n\n", frame_index + 1);
    printf(1,"%s", frames[frame_index]);
}

int main(int argc, char *argv[]) {
    int num_frames = sizeof(frames) / sizeof(frames[0]);
    int delay = 10;  // Delay between frames in ticks

    for (int i = 0; i < 40; i++) {  // Show 20 frames in total
        draw_frame(i % num_frames);
        sleep(delay);
    }
    exit();
}
```

3. The necessary header files are included first.

4. Then we create a clear_screen() function which uses the clear() system call which we created. Clear system call is used to clear the console. Steps to create this call are shown later.

5. The draw_frame() function, clears the console and then prints a frame of the animation(decided by the frame_index parameter)

6. Finally, the main function runs the draw_frame() function in a loop, displaying a total of 40 frames, each after a delay of 10 ticks.

Finally we can run the animation user program to show an animation of a launching rocket!

We followed these steps to create the “clear” system call:

1. **syscall.h** - Added the system call SYS_clear

```c
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_clear  22
```

2. **syscall.c** - Added the following

syscall.c × syscall.h

Ubuntu > home > shashwat > xv6-public > syscall.c

```c
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_clear]   sys_clear,
};
```

syscall.c × syscall.h

Ubuntu > home > shashwat > xv6-public > syscall.c

```c
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_clear(void);
```

3. **user.h** -

user.h ×

Ubuntu > home > shashwat > xv6-public > user.h

```c
char* sbrk(int);
int sleep(int);
int uptime(void);
int clear(void);
```

4. **usys.S -** Now we want the user to be able to use the system call. We added the line ‘SYSCALL(clear)’

usys.S ×

Ubuntu > home > shashwat > xv6-public > usys.S

```
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(clear)
```

5. **sysclear.c** - Created a new file sysclear.c which calls the console clear function

sysclear.c ×

Ubuntu > home > shashwat > xv6-public > sysclear.c

```c
#include "types.h"
#include "defs.h"
#include "param.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "fs.h"
#include "file.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "console.h"

extern void consoleclear(void);

int
sys_clear(void)
{
  consoleclear();
  return 0;
}
```

6. **console.c** - created a new function console clear which clears the console after printing each frame.

7. Added the prototype "void consoleclear(void)" in **console.h** and **defs.h**

8. In **Makefile**, we added sysclear.o to the OBJS list.

console.c

Ubuntu > home > shashwat > xv6-public > console.c

```c
// #define INPUT_BUF 128
#define CRT_ROWS 25
#define CRT_COLS 80
#define CRT_SIZE (CRT_ROWS * CRT_COLS)

#define CRT_WHITE 0x07
#define CRT_CURSOR_H 14
#define CRT_CURSOR_L 15

void
consoleclear(void)
{
  int i;

  acquire(&cons.lock);

  // Move cursor to top-left
  outb(CRTPORT, 14);
  outb(CRTPORT+1, 0);
  outb(CRTPORT, 15);
  outb(CRTPORT+1, 0);

  // Fill screen with spaces
  for(i = 0; i < CRT_SIZE; i++) {
    crt[i] = (CRT_WHITE << 8) | ' ';
  }

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

<u>**Testing the Program:**</u> Run the program by calling the animation function

This is the video.

# Task-1.3

**Files changed:**

1. **Proc.h -** Add these variable in proc struct

- ctime: Creation time of the process.
- stime: Time spent in the SLEEPING state.
- retime: Time spent in the READY (RUNNABLE) state.
- rutime: Time spent in the RUNNING state.

2. **Proc.c -** We ensured that process creation initializes ctime and updates stime, retime, and rutime correctly during context switches and clock ticks. The scheduler logic was modified to handle updates to these variables. However, while stime and retime are updated correctly, rutime is not, potentially due to improper handling of state transitions or clock tick updates for rutime.

proc.c

Ubuntu > home > shashwat > xv6-public > proc.c

```c
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  p->ctime = ticks;
  p->stime = 0;
  p->retime = 0;
  p->rutime = 0;

  release(&ptable.lock);
```

proc.c

Ubuntu > home > shashwat > xv6-public > proc.c

```c
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

   acquire(&ptable.lock);
   // loop through process to increment the variables
    for(p = ptable.proc; p<&ptable.proc[NPROC];p++){
      if(p->state == RUNNABLE)
        p->retime++; // increment retime for the processes that are ready to run
      else if(p->state == RUNNING)
        p->rutime++;// increment rutime for the processes that are running
      else if(p->state == SLEEPING)
        p->stime++; // increment stime for the processes that are sleeping
    }

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE) continue;

      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
```

3. **user.h -** declare the new system call int wait2(int *retime, int *rutime, int *stime);

```c
int uptime(void);
int clear(void);
int wait2(int *retime, int *rutime, int *stime);
```

4. **Syscall.h -** Added “SYS_wait2 23”

```c
#define SYS_close  21
#define SYS_clear  22
#define SYS_wait2  23
```

5. **syscall.c** - Implement wait2 to collect and return the time metrics for a terminated child process.

```c
[SYS_close]   sys_close,
[SYS_clear]   sys_clear,
[SYS_wait2]   sys_wait2,
};
```

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

6. **sysproc.c** - Ensure proper integration of the new system call, and verify that it returns the correct values.

```c
int wait2(int *retime, int *rutime, int *stime);

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

7. **Trap.c** - Modify trap.c to accurately increment rutime during process execution. While rutime is updated correctly, note that stime and retime remain unaffected, possibly due to incomplete or incorrect state management.

trap.c

Ubuntu > home > shashwat > xv6-public > trap.c

```c
//PAGEBREAK: 41
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:
    if(myproc()!=0 && myproc()->state == RUNNING){
      myproc()->rutime++;
    }
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
```

# Test Cases:

- **Test Case 1**: This test case is designed to simulate a process that alternates between CPU-bound work and I/O-bound sleep periods. It tests whether the wait2 system call accurately tracks time spent in both the running state (rutime) and the sleeping state (stime), along with time spent ready to run (retime).

**Child Process Behavior:**

The child process runs a loop 5 times, where it alternates between:

- CPU Work: Performs a CPU-bound operation using a loop (for (volatile int i = 0; i < 500000000; i++);). This simulates significant computational activity, which should increase the rutime.
- Sleeping: Calls sleep(50); after each CPU task. This simulates waiting for I/O operations, which should increase the stime.

The process switches back and forth between CPU work and sleeping, providing a mix of both behaviors.

**Parent Process Behavior:**

The parent process waits for the child to finish using wait2(&retime, &rutime, &stime);

It then prints out the PID of the child process and the values of retime, rutime, and stime.

```
// Test Case 1 - Alternating CPU-bound and I/O-bound Process
if ((pid = fork()) == 0) {
    // Child process: Alternates between CPU-bound and I/O-bound tasks
    for (int j = 0; j < 5; j++) {
        // CPU-bound task
        for (volatile int i = 0; i < 200000000; i++);
        // I/O-bound task (sleep)
        sleep(20);
    }
    exit();
} else {
    // Parent process waits for child
    wait2(&retime, &rutime, &stime);
    printf(1, "Test Case 1 - Alternating CPU-bound and I/O-bound Process\n");
    printf(1, "Child PID: %d\n", pid);
    printf(1, "Retime: %d\n", retime);
    printf(1, "Rutime: %d\n", rutime);
    printf(1, "Stime: %d\n", stime);
}
```

- **Test Case 2:** This test case is designed to evaluate how the operating system handles a process that is predominantly I/O-bound with minimal CPU activity. It checks whether the wait2 system call accurately tracks the amount of time a process spends in different states: sleeping (I/O-bound), running (CPU-bound), and ready.

**Child Process Behavior:**

Sleeping: The child process sleeps for a long duration in each of 10 iterations (sleep(50);). This simulates waiting for I/O and should increase the stime value.

Minimal CPU Work: After sleeping, the process performs a very short CPU-bound task (for (volatile int j = 0; j < 1000000; j++);), which minimally increases rutime.

**Parent Process Behavior:**

The parent process waits for the child to complete using wait2(&retime, &rutime, &stime);.

It then prints the PID of the child along with the recorded retime, rutime, and stime.

```
// Test Case 2 - Long I/O-bound Process
if ((pid = fork()) == 0) {
    // Child process: Mostly I/O-bound with little CPU work
    for (int i = 0; i < 10; i++) {
        sleep(50);  // Sleep for a long duration
    }
    for (volatile int j = 0; j < 1000000; j++);  // Minimal CPU work
    exit();
} else {
    // Parent process waits for child
    wait2(&retime, &rutime, &stime);
    printf(1, "Test Case 2 - Long I/O-bound Process\n");
    printf(1, "Child PID: %d\n", pid);
    printf(1, "Retime: %d\n", retime);
    printf(1, "Rutime: %d\n", rutime);
    printf(1, "Stime: %d\n", stime);
}
```

- **Test Case 3:** This test case aims to simulate a process that rapidly alternates between CPU-bound work and sleeping. It tests whether the wait2 system call can accurately track frequent state transitions between running, sleeping, and ready states.

**Child Process Behavior:**

The child process runs a loop 20 times, alternating between:

CPU Work: It performs a short CPU-bound task (for (volatile int j = 0; j < 10000000; j++);). This increases rutime as the process spends time actively using the CPU.

Sleeping: It then sleeps for a short duration (sleep(5);). This simulates waiting for I/O and should increase the stime value.

This frequent switching between running and sleeping creates multiple state transitions.

**Parent Process Behavior:**

The parent process waits for the child to complete using wait2(&retime, &rutime, &stime);.

It prints the PID of the child along with the recorded retime, rutime, and stime.

```c
// Test Case 3 - Frequent State Changes
if ((pid = fork()) == 0) {
    // Child process: Alternates rapidly between CPU and I/O-bound
    for (int i = 0; i < 20; i++) {
        for (volatile int j = 0; j < 10000000; j++); // Short CPU work
        sleep(5);  // Short sleep
    }
    exit();
} else {
    // Parent process waits for child
    wait2(&retime, &rutime, &stime);
    printf(1, "Test Case 3 - Frequent State Changes\n");
    printf(1, "Child PID: %d\n", pid);
    printf(1, "Retime: %d\n", retime);
    printf(1, "Rutime: %d\n", rutime);
    printf(1, "Stime: %d\n", stime);
}
```

- **Test Case 4:** This test case is designed to simulate a process that spends most of its time in the sleeping state, with very little to no CPU work. It tests whether the wait2 system call accurately tracks processes that are almost entirely I/O-bound.

**Child Process Behavior:**

The child process executes a loop that runs 50 times, where it:

Sleeping: Calls sleep(10); in each iteration, which makes the process spend a significant amount of time waiting (simulating I/O-bound behavior).

There is no additional CPU work, so the process is mostly in the sleeping state.

**Parent Process Behavior:**

The parent process waits for the child to finish using wait2(&retime, &rutime, &stime);.

It then prints out the PID of the child process along with the values of retime, rutime, and stime.

```c
// Test Case 4 - Multiple Sleep Calls
if ((pid = fork()) == 0) {
    // Child process: Mostly sleeping
    for (int i = 0; i < 50; i++) {
        sleep(10);  // Sleep multiple times
    }
    exit();
} else {
    // Parent process waits for child
    wait2(&retime, &rutime, &stime);
    printf(1, "Test Case 4 - Multiple Sleep Calls\n");
    printf(1, "Child PID: %d\n", pid);
    printf(1, "Retime: %d\n", retime);
    printf(1, "Rutime: %d\n", rutime);
    printf(1, "Stime: %d\n", stime);
}

int retime1, rutime1, stime1;
int retime2, rutime2, stime2;
int retime3, rutime3, stime3;
int pid1, pid2, pid3;
```

- **<u>Test Case 5:</u>** This test case is designed to simulate a process that is primarily CPU-bound with only minimal periods of sleeping. It tests whether the wait2 system call accurately records the run time (rutime) for processes that engage mostly in CPU-intensive work.

**<u>Child Process Behavior:</u>**

The child process runs a loop 5 times, where it:

CPU Work: Performs intensive CPU-bound operations using a loop (for (volatile int j = 0; j < 300000000; j++);). This consumes significant processing time, increasing rutime.

Minimal Sleeping: It then calls sleep(2); after each CPU task, introducing minimal sleep periods. This slight sleep is meant to simulate very brief I/O waits.

The combination ensures that the process is mainly using the CPU, with only short interruptions for sleep.

**<u>Parent Process Behavior:</u>**

The parent process waits for the child to complete using wait2(&retime, &rutime, &stime);

It prints the PID of the child process along with the collected retime, rutime, and stime.

```c
// Test Case 5 - Multiple Concurrent Processes with Mixed Behavior

// Fork the first child process (CPU-bound)
if ((pid1 = fork()) == 0) {
    // Child 1: CPU-bound task
    for (volatile int i = 0; i < 500000000; i++);  // Intensive CPU work
    exit();
}

// Fork the second child process (I/O-bound)
if ((pid2 = fork()) == 0) {
    // Child 2: I/O-bound task
    for (int i = 0; i < 10; i++) {
        sleep(20);  // Sleep multiple times to simulate I/O
    }
    exit();
}

// Fork the third child process (Mixed CPU-bound and I/O-bound)
if ((pid3 = fork()) == 0) {
    // Child 3: Mixed behavior
    for (int i = 0; i < 5; i++) {
        for (volatile int j = 0; j < 100000000; j++);  // Some CPU work
        sleep(10);  // Sleep to simulate I/O wait
    }
    exit();
}

// Parent process waits for all three children and prints their statistics

// Wait for the first child (CPU-bound)
wait2(&retime1, &rutime1, &stime1);
printf(1, "Test Case 5 - Multiple Concurrent Processes with Mixed Behavior\n");
printf(1, "Child 1 (CPU-bound) PID: %d\n", pid1);
printf(1, "Retime: %d\n", retime1);
printf(1, "Rutime: %d\n", rutime1);
printf(1, "Stime: %d\n", stime1);

// Wait for the second child (I/O-bound)
wait2(&retime2, &rutime2, &stime2);
printf(1, "Child 2 (I/O-bound) PID: %d\n", pid2);
printf(1, "Retime: %d\n", retime2);
printf(1, "Rutime: %d\n", rutime2);
printf(1, "Stime: %d\n", stime2);

// Wait for the third child (Mixed CPU-bound and I/O-bound)
wait2(&retime3, &rutime3, &stime3);
printf(1, "Child 3 (Mixed behavior) PID: %d\n", pid3);
printf(1, "Retime: %d\n", retime3);
printf(1, "Rutime: %d\n", rutime3);
printf(1, "Stime: %d\n", stime3);
```

Output of the testcases:

```
$ task3
Test Case 1 - Alternating CPU-bound and I/O-bound Process
Child PID: 4
Retime: 355
Rutime: 255
Stime: 403055
Test Case 2 - Long I/O-bound Process
Child PID: 5
Retime: 500
Rutime: 0
Stime: 1713891
Test Case 3 - Frequent State Changes
Child PID: 6
Retime: 151
Rutime: 51
Stime: 265703
Test Case 4 - Multiple Sleep Calls
Child PID: 7
Retime: 501
Rutime: 1
Stime: 1613235
Test Case 5 - Multiple Concurrent Processes with Mixed Behavior
Child 1 (CPU-bound) PID: 8
Retime: 85
Rutime: 0
Stime: 30
Child 2 (I/O-bound) PID: 9
Retime: 137
Rutime: 137
Stime: 0
Child 3 (Mixed behavior) PID: 10
Retime: 153
Rutime: 133
Stime: 73022
$
```