

# **GROUP 9 MNC**

## **Team Members :**

- 1.Srinjoy Som**
- 2.Shubham Kumar Jha**
- 3.Chandrashekhar Dharmarajan**
- 4.Dhruva Nilesch Kuthari**

## **Drive Folder for patch file and required code :**

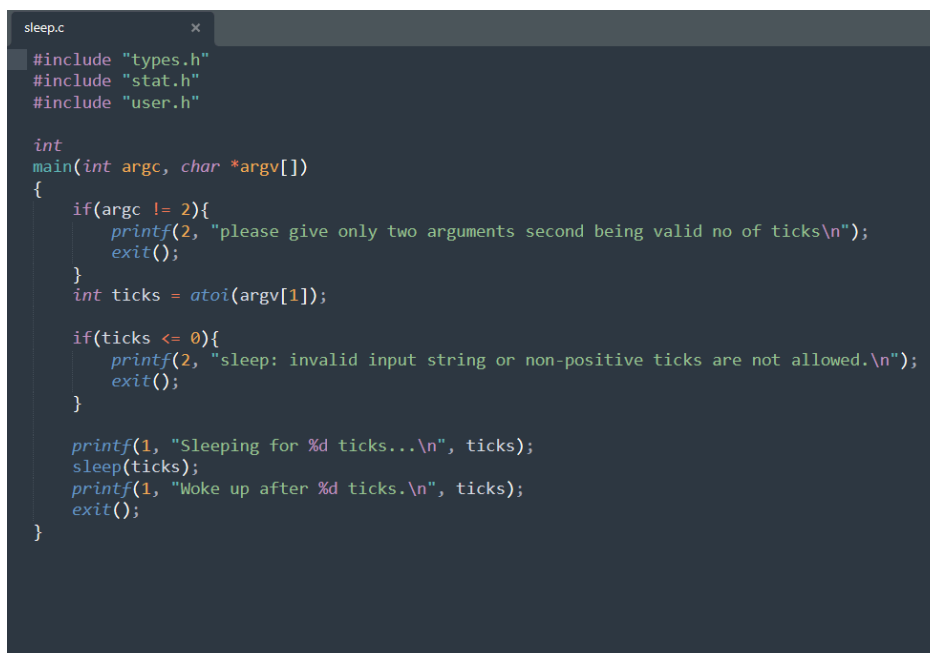
<https://drive.google.com/drive/folders/18TdWWQ3YIJRisG84tJrQXonlkrkBlvfa?usp=sharing>

# Assignment 1

## QUESTION 1.1

**Objective:** We wanted to create a user program which takes in the user input of the number of ticks (via command line argument) and puts the program to sleep for the specified timeframe

We used the sleep system call used in the wait system call to implement the following user level program.



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

int
main(int argc, char *argv[])
{
    if(argc != 2){
        printf(2, "please give only two arguments second being valid no of ticks\n");
        exit();
    }
    int ticks = atoi(argv[1]);

    if(ticks <= 0){
        printf(2, "sleep: invalid input string or non-positive ticks are not allowed.\n");
        exit();
    }

    printf(1, "Sleeping for %d ticks...\n", ticks);
    sleep(ticks);
    printf(1, "Woke up after %d ticks.\n", ticks);
    exit();
}
```

C code for sleep.c program.

### **Implementation :**

1. In this user program we need argc =2 (one is number of ticks, and the first one being the name of the user program ). Otherwise, the program gives an error stating the issue.
2. If the number of ticks is negative we consider it has invalid input and print that out in the stdout file .
3. If no of ticks is a positive number, we use the already existing sleep system call to sleep for that many number of ticks.
4. Finally we included this file in UPROGS section of MAKEFILE so that it gets compiled along when the OS boots up next time .

```

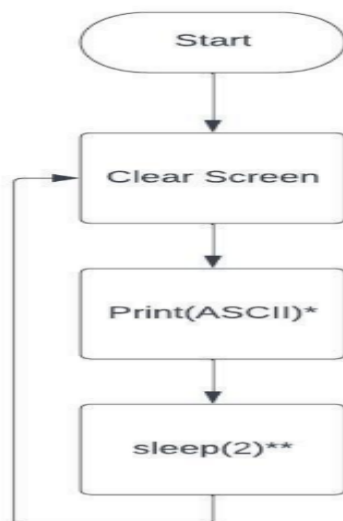
6 $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o _forktest forktest.o ulib.o usys.o
7 $(OBJDUMP) -S _forktest > forktest.asm
8
9 mkfs: mkfs.c fs.h
10 gcc -Werror -Wall -o mkfs mkfs.c
11
12 # Prevent deletion of intermediate files, e.g. cat.o, after first build, so
13 # that disk image changes after first build are persistent until clean. More
14 # details:
15 # http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
16 .PRECIOUS: %.o
17
18 UPROGS=\
19 _cat\
20 _echo\
21 _forktest\
22 _grep\
23 _init\
24 _kill\
25 _ln\
26 _ls\
27 _mkdir\
28 _sleep\
29 _animation\
30 _rm\
31 _sh\
32 _stressfs\
33 _usertests\
34 _wc\
35 _zombie\

```

Adding sleep in UPROGS

## QUESTION 1.2

**Objective:** To create a user program which provides a user interface according to the flowchart shown below .



So the bash should print out an ASCII art form , following that the process should sleep for a required amount of ticks and then the screen should be cleared . This procedure should continue in a loop . It creates a blinking effect of the image .



## QUESTION 1.3:

### **Objective :**

We wanted to create a system call wait2 which is extension of wait system call. It takes as arguments the time for which a process was in SLEEPING, READY(RUNNABLE) and RUNNING state, updates them and returns the PID of terminated child process (if successful) or 1 (upon failure).

First we update the struct proc to include the ctime, stime, retime, runtime. We do the following change in proc.h.

**1 .proc.h:** Extend the **proc** struct by adding the following fields:

- i) **ctime**: creation time of the process
- ii) **stime**: time for which the process was sleeping
- iii) **retime**: time for which the process was ready
- iv) **runtime**: time for which the process was 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; // switch() 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 ctime; // Creation time
    int stime; // Sleep time (time in SLEEPING state)
    int retime; // Ready time (time in READY/RUNNABLE state)
    int runtime; // Run time (time in RUNNING state)
};
```

2. Now in proc.c we change the allocproc function so that whenever a process is allocated (via fork), then appropriate values for these variables are initialised :

**Proc.c:**

In allocproc we initialise the values of these fields.

- i) **ctime** = ticks

- ii) **stime** = 0
- iii) **retime** = 0
- iv) **runtime** = 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++;

    // Initialize the new time fields
    p->ctime = ticks; // Set creation time to current ticks
    p->stime = 0;
    p->retime = 0;
    p->rtime = 0;

    release(&ptable.lock);

    // Allocate kernel stack.
    if((p->kstack = kalloc()) == 0){
        p->state = UNUSED;
        return 0;
    }
    sp = p->kstack + KSTACKSIZE;

    // Leave room for trap frame.
    sp -= sizeof *p->tf;
    p->tf = (struct trapframe*)sp;

    // Set up new context to start executing at forkret,
    // which returns to trapret.
    sp -= 4;
    *(uint*)sp = (uint)trapret;

    sp -= sizeof *p->context;
    p->context = (struct context*)sp;
    memset(p->context, 0, sizeof *p->context);
    p->context->eip = (uint)forkret;

    return p;
}
```

Now the approach is to use the C trap function to update these parameters. Whenever the CPU will generate the timer interrupt the trap function will be called and we will use this opportunity to update the values for all processes in the ptable. For that we have written an update\_value function, which is called in the C trap function.

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
        acquire(&tickslock);
        ticks++;

        // calling the updateValues function
        updateValues();

        wakeup(&ticks);
        release(&tickslock);
    }

    lapiceoi();
    break;
}
```

```
void
updateValues()
{
    struct proc *p;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state == SLEEPING){
            p->stime++;
        }
        else if(p->state == RUNNABLE){
            p->retime++;
        }
        else if(p->state == RUNNING){
            p->rtime++;
        }
    }
    release(&ptable.lock);
}
```

The trap function has been updated in trap.c and the upateValues has been updates in proc.c . Appropriate changes has been made in def.h so that trap.c can access update\_values .

We implement the wait2 function which returns the requires values. This wait2 function will be used by the kernel system call SYS\_wait2 . Now the parent process can use this system call to access the values for its child process . This code has been updated in proc.c

```
int
wait2(int *retime, int *rtime, 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;
            if (p->state == ZOMBIE) {
                if(copyout(curproc->pgdir, (uint)retime, (char *)&p->retime, sizeof(int)) < 0 ||
                    copyout(curproc->pgdir, (uint)rtime, (char *)&p->rtime, sizeof(int)) < 0 ||
                    copyout(curproc->pgdir, (uint)stime, (char *)&p->stime, sizeof(int)) < 0) {
                        release(&ptable.lock);
                        return -1;
                    }
                // Found a zombie process
                pid = p->pid;

                // Clean up the process
                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;
                p->ctime=0;
                p->stime=0;
                p->retime=0;
                p->rtime=0;

                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.
    sleep(curproc, &ptable.lock); // DOC: wait-sleep
}
```

Now we implement the SYS\_wait2 system call using a similar process as we did in Assignment 0B1 :

- **user.h** : To provide user with the syscall library.
- **usys.S** : To create a trap macro for wait2.
- **proc.c** : To implement the wait2 function for ptable access.
- **syscall.c** : To make the syscall function route to our custom system call.
- **Syscall.h** : Add system call number for wait2

```
int sleep(int);  
int uptime(void);  
int draw(void *, uint);  
int wait2(int*, int*, int*);
```

User.h

```
SYSCALL(sleep)  
SYSCALL(uptime)  
SYSCALL(draw)  
SYSCALL(wait2)
```

Usys.S

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

```
[SYS_mkdir] sys_mkdir,  
[SYS_close] sys_close,  
[SYS_draw] sys_draw,  
[SYS_wait2] sys_wait2,  
};
```

Syscall.c

```
#define SYS_mkdir 20  
#define SYS_close 21  
#define SYS_draw 22  
#define SYS_wait2 23
```

Syscall.h



```

int
sys_wait2(void)
{
    int *retime, *rtime, *stime;

    if (argptr(0, (void*)&retime, sizeof(retime)) < 0)
        return -1;
    if (argptr(1, (void*)&rtime, sizeof(rtime)) < 0)
        return -1;
    if (argptr(2, (void*)&stime, sizeof(stime)) < 0)
        return -1;

    return wait2(retime, rtime, stime);
}

```

**System call defined in sysproc.c**

### **Testing our system Call**

Now we will write a user program to test our code . The user test program is called wait2.c . Note this is not the wait2 function defined above . It is the wait2 user program that tests the above defined system call . Below we give the description of what the program does .

Wait 2 system call is then called and if the returned value is -1, which means no terminated child process is found.

If the returned value is some positive value then it is PID of the child process which has terminated. Along with this during this call the value of stime, retime, and rtime have been updated. So we print the returned PID, stime, retime and rtime.

C wait2.c U X

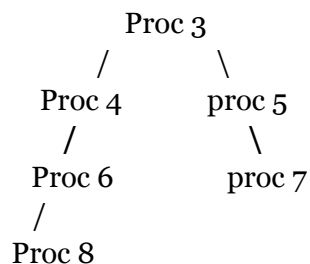
xv6\_mod > C wait2.c > ...

```
11 int main() {
12     int retime, rtime, stime;
13     int pid = fork();
14     int pid2= fork();
15     if(pid<0){
16         printf(1, "Fork failed");
17     }
18     else if (pid == 0) {
19         // Child process
20         volatile int j =100000000;
21         while(j>0) {
22             int annns=ret_pow(2 , 10);
23             annns=annns*0;
24             j--;
25         }
26         sleep(20);
27     } else {
28         int ans = wait2(&retime, &rtime, &stime);
29         if (ans > 0) {
30             printf(1, "child with pid: %d in parent %d \n", ans , (int)getpid() );
31             printf(1, "for pid: %d Wait2 returned: retime=%d, rtime=%d, stime=%d\n", ans , retime, rtime, stime);
32         } else {
33             printf(1, "in parent %d wait2 failed for child %d \n" ,(int)getpid() , ans );
34         }
35     }
36
37     if(pid2<0){
38         printf(1, "Fork failed");
39     }
40     else if (pid2 == 0) {
41         // Child process
42         volatile int j =100000000;
43         while(j>0) {
44             int annns=ret_pow(2 , 10);
45             annns=annns*0;
46             j--;
47         }
48         sleep(20);
49         //child of child
50         int pid3=fork();
51         if(pid3<0){
52             int pid3=fork();
53             if(pid3<0){
54                 printf(1, "Fork failed");
55             }
56             else if (pid3 == 0) {
57                 // Child process
58                 volatile int j =100000000;
59                 while(j>0) {
60                     int annns=ret_pow(2 , 10);
61                     annns=annns*0;
62                     j--;
63                 }
64                 sleep(20);
65             } else {
66                 int ans = wait2(&retime, &rtime, &stime);
67                 if (ans > 0) {
68                     printf(1, "child with pid: %d in parent %d \n", ans , (int)getpid() );
69                     printf(1, "for process %d Wait2 returned: retime=%d, rtime=%d, stime=%d\n", ans , retime, rtime, stime);
70                 } else {
71                     printf(1, "in parent %d wait2 failed for child %d \n" ,(int)getpid() , ans );
72                 }
73             }
74         } else {
75             int ans = wait2(&retime, &rtime, &stime);
76             if (ans > 0) {
77                 printf(1, "child is pid: %d in parent %d \n", ans , (int)getpid() );
78                 printf(1, "for process %d Wait2 returned: retime=%d, rtime=%d, stime=%d\n", ans , retime, rtime, stime);
79             } else {
80                 printf(1, "in parent %d wait2 failed for child %d \n" ,(int)getpid() , ans );
81             }
82         }
83     }
84     exit();
85 }
```

## Output of the code and explanation

```
enter: starting sh
$ wait2
in parent 5 wait2 failed
child with pid: 7 in parent 5
for process 7 Wait2 returned: retime=46, rtime=47, stime=20
child with pid: 5 in parent 3
for pid: 5 Wait2 returned: retime=95, rtime=47, stime=133
child with pid: 8 in parent 6
for process 8 Wait2 returned: retime=0, rtime=47, stime=20
child is pid: 6 in parent 4
for process 6 Wait2 returned: retime=140, rtime=97, stime=107
child is pid: 4 in parent 3
for process 4 Wait2 returned: retime=95, rtime=49, stime=201
$ █
```

## Digramatic explanation



Further we added this file in UPROGS so that it can be compiled with the operating system .

We also have another usertest called testwait2.c which solidifies the proof that the system call is functioning smoothly .

```

int main() {
for(int i=0;i<2;i++){
    int retime, rtime, stime;
    int pid = fork();
    if(pid<0){
        printf(1, "Fork failed");
    }
    else if (pid == 0) {
        // Child process
        printf(1, "inside child process\n");
        volatile int j =100000000;
        while(j>0) {
            asm("");
            int annns=ret_pow(2 , 10);
            annns=annns*0;
            j--;
        }
        sleep(20);
    } else {
        asm("");
        int ans = wait2(&retime, &rtime, &stime);
        if (ans > 0) {
            printf(1, "pid: %d\n", ans);
            printf(1, "Wait2 returned: retime=%d, rtime=%d, stime=%d\n", retime, rtime, stime);
        } else {
            printf(1, "wait2 failed\n");
        }
    }
}
}
exit();
}

```

## Testwait2.c

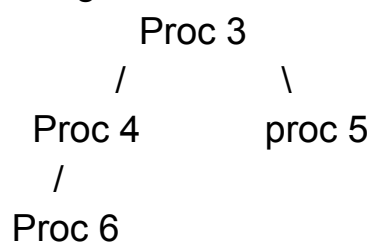
### Output and Explanation

```

$ testwait2
inside child process
inside child process
pid: 5
Wait2 returned: retime=0, rtime=52, stime=20
pid: 4
Wait2 returned: retime=0, rtime=50, stime=92
inside child process
pid: 6
Wait2 returned: retime=0, rtime=52, stime=20
$

```

With process Diagram



**Explanation:** In the second case processes 5 takes 20 ticks of sleep as we hardcoded and runs for 52 ticks in cpu similar to process 6 .process 4 runs for 50 ticks and waits 20 ticks of itself and 52 ticks when proc 6 is running in cpu and 20 ticks when proc 6 is sleeping giving total 92 ticks of sleep time.