Q2. First of all we updated the proc.h file where we added **ctime, retime, rutime and stime** inside the proc struct. After that we defined the functions **statisitics_handler(Will update the process time values depending upon it's state) and wait2(Extends the wait system call)** inside the same file.

```
38  struct proc {
39    uint sz;                     // Size of process memory (bytes)
40    pde_t* pgdir;                // Page table
41    char *kstack;                // Bottom of kernel stack for this process
42    enum procstate state;        // Process state
43    int pid;                     // Process ID
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscall
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52    int ctime;                   // Time of creation
53    int stime;                   //  Number of clock ticks during which the process was waiting for I/O
54    int retime;                  //  Number of clock ticks during which the process waited
55    int rutime;                  //  Number of clock ticks during which the process was running
56  };
57  // Process memory is laid out contiguously, low addresses first:
58  //   text
59  //   original data and bss
60  //   fixed-size stack
61  //   expandable heap
62
63  void statistics_handler();      // Will update the process time values depending upon it's state
64  int  wait2(int*, int*, int*);   // Extends the wait system call
```

proc.h

Then we updated the **proc.c** file where we implemented the **wait2 and statistics_handler** functions. Also we updated the **allocproc** function.

int wait2(int* retime,int *rutime, int*stime):-

This function takes 3 arguments i.e. retime, rutime and stime and updates these values inside it. It returns the PID of the successfully terminated child process if it is not possible then it will return 1. In this function we are first checking where the current process is the parent of the process currently being searched. If it is so then we are checking whether the process is in ZOMBIE or not. If it is then we are terminating that process and assigning corresponding time statistics values inside the arguments taken and then finally return the PID of the process.

```
329  int wait2(int *retime, int *rutime, int *stime)
330  {
331    struct proc *p;
332    int havekids, pid;
333    struct proc *curproc = myproc();
334
335    acquire(&ptable.lock);
336    for (;;)
337    {
338      // Scan through table looking for exited children.
339      havekids = 0;
340      for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
341      {
342        if (p->parent != curproc)
343          continue;
344        havekids = 1;
345        if (p->state == ZOMBIE)
346        {
347          // Found one.
348          pid = p->pid;
349          *retime=p->retime;
350          *stime=p->stime;
351          *rutime=p->rutime;
352
353          kfree(p->kstack);
354          p->kstack = 0;
355          freevm(p->pgdir);
356          p->pid = 0;
357          p->parent = 0;
358          p->name[0] = 0;
359          p->killed = 0;
360
361          p->ctime = 0;
362          p->retime = 0;
363          p->rutime = 0;
364          p->stime = 0;
365
366          p->state = UNUSED;
367          release(&ptable.lock);
368          return pid;
369        }
370      }
371
372      // No point waiting if we don't have any children.
373      if (!havekids || curproc->killed)
374      {
375        release(&ptable.lock);
376        return 1;
377      }
378
379      // Wait for children to exit.  (See wakeup1 call in proc_exit.)
380      sleep(curproc, &ptable.lock); //DOC: wait-sleep
381    }
382  }
383
```

void statistics_handler() :-

In this function we will iterate through each process present inside the process table and will update the process according to the state(RUNNING, SLEEPING, RUNNABLE) in which it is present.

```
void statistics_handler()
{
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    switch (p->state)
    {
    case SLEEPING:
      p->stime++;
      break;
    case RUNNABLE:
      p->retime++;
      break;
    case RUNNING:
      p->rutime++;
      break;
    default:;
    }
  }
  release(&ptable.lock);
}
```

static struct proc* allocproc(void)

Here as a new process is getting created therefore we are assigning **ctime=ticks** and **retime, rutime, stime** as zero.

```
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->rutime = 0;
  p->retime = 0;
  p->stime = 0;
  release(&ptable.lock);

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

Then we updated the **trap.c** file. Inside the trap function we called the **statistics_handler** inside the switch in the case where tf->trapno is equal to T_IRQ +IRQ_TIMER. Whenever ticks gets updated i.e. incremented we are also calling the statistics_handler that time which will update all the processes available.

```
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(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      statistics_handler();
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case T_IRQ0 + IRQ_KBD:
```

Then I created a system call which will be used to call the wait2 function inside the **sysproc.c** file. In this we are first defining 3 integer pointer variables and placing values inside them (the one received from the user). Then we are calling the wait2 function and then returning the value wait2 returned.

```
93   int sys_wait2(void) {
94     int *retime, *rutime, *stime;
95     if (argptr(0, (void*)&retime, sizeof(retime)) < 0)
96       return -1;
97     if (argptr(1, (void*)&rutime, sizeof(retime)) < 0)
98       return -1;
99     if (argptr(2, (void*)&stime, sizeof(stime)) < 0)
100      return -1;
101    return wait2(retime, rutime, stime);
102  }
103
```

After that we defined a macro for this function and gave it the system call number 22 inside the **syscall.h** file and this helped us to connect it to the implemented function.

```c
// 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_wait2  22
```

Then to make this function visible to the entire program I added the line **extern int sys_wait2(void)** inside the **syscall.c** file and thus it connects the shell and the kernel. Then inside the syscalls array we added sys_wait2 at 22nd position.

```c
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_wait2(void);

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_wait2]   sys_wait2,
};
```

Then to connect the user's call to the system call function we created a user level system call definition inside the **usys.S** file.

usys.S

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(wait2)
```

Then comes the final part, inside the **user.h** (user header file) I added the user call **int wait2(int*, int*, int*)** . And this user call is the one which takes 3 arguments retime, rutime, stime and updates them and then finally returns the PID of the terminated child process.

```
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int wait2(int *,int *,int *);
```

Then we created a user level test file named **wait2_test.c** which calls this function and prints the respective information. In this file we are first creating new duplicate child processes by using **fork()** and then calling wait2 and then storing it's value in the pid variable. Then if the fork returns 0 value then it means that the current process is a child process. Therefore for that iteration we are just

printing “Child Process” in the console. If it is not zero then we will check whether pid==fork’s return value; if it so then we will print the retime,rutime and stime as it returns the terminated child process. Else we will print the error that is not able to terminate the child process on the console.

```c
C wait2_test.c > main()
#include "types.h"
#include "user.h"

int main()
{
    int retime, rutime, stime, n1;
    for (int a = 0; a < 3; a++)
    {
        n1 = fork();
        int pid = wait2(&retime, &rutime, &stime);
        if (!n1)
        {
            printf(1, "Child Process.\n");
            continue;
        }
        if (pid == n1)
        {
            printf(1, "| PID: %d | Retime: %d | Rutime: %d | Stime: %d |\n", n1, retime, rutime, stime);
        }
        else
        {
            printf(1, "Not able to terminate the child process.\n");
        }
    }
    exit();
}
```

```
$ wait2_test
Child Process.
Child Process.
Child Process.
| PID: 6 | Retime: 0 | Rutime: 0 | Stime: 0 |
| PID: 5 | Retime: 0 | Rutime: 3 | Stime: 0 |
Child Process.
| PID: 7 | Retime: 0 | Rutime: 1 | Stime: 0 |
| PID: 4 | Retime: 0 | Rutime: 4 | Stime: 4 |
Child Process.
Child Process.
| PID: 9 | Retime: 0 | Rutime: 1 | Stime: 0 |
| PID: 8 | Retime: 0 | Rutime: 4 | Stime: 1 |
Child Process.
| PID: 10 | Retime: 0 | Rutime: 1 | Stime: 0 |
$
```

The above picture depicts the output of the file. As we can see there are 7 child processes present and therefore we can terminate 7 child processes whose statistics are also shown.

So to make this program available for xv6 source code for compilation we included wait2_test in the User programs or UPROGS by adding the line **_wait2_test** inside the **UPROGS** present inside the **Makefile.**

Then we reloaded the QEMU terminal by executing following commands:-

1.) make clean
2.) make qemu

These commands will clean, compile and link things respectively.

Then I executed the **wait2_test** to get the desired output. We can also list all the **fs.img** content using ls command through which we ensure that the wait2_test command is available.