

Operating Systems

Assignment 2 - Lab Report

Group 4

Link to patch files :

<https://drive.google.com/drive/folders/1TK9tdpd0nxJU6WZ-AYTUoLmb50hhFow0?usp=sharing>

Group Members:

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

PART A:

Adding the system call settickets(number) :

Adding the function declaration in **user.h** :

```
int settickets(int number);
```

Declaring the system call in **usys.S** :

```
SYSCALL(sys_settickets)
```

Now, the following code was added in **sysproc.c** to carry out the required functionality:

```
int sys_settickets(void) {  
    int number;  
    if(argint(0, &number) < 0 || number <= 0 || number > 100000)  
        return -1;  
    myproc()->tickets = number;  
    return 0;  
}
```

The code checks if the argument is invalid and returns -1 accordingly. The code then assigns the specified number of tickets to the current process.

A system call number was allotted to this system call in **syscall.h** :

```
#define SYS_close 21  
#define SYS_settickets 22
```

The system call was added in the array of function pointers and declared in **syscall.c** :

```
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_settickets] sys_settickets,  
}
```

```
extern int sys_settickets(void);
```

Adding system call getprocessinfo(struct processes_info *p) :

First, struct proc was modified to keep track of tickets and ticks of each process:

```
// 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 tickets;
    int ticks;
};
```

The struct `processes_info` was created :

```
struct processes_info {
    int num_processes;
    int pids[NPROC];
    int ticks[NPROC];
    int tickets[NPROC];
};
```

The corresponding declarations were added in **syscall.c** :

```
extern int sys_settickets(void);
extern int sys_getprocessesinfo(void);

[sys_settickets] sys_settickets,
[SYS_getprocessesinfo] sys_getprocessesinfo,
[SYS_gettickets] sys_gettickets,
```

A system call number was allotted to this system call in **syscall.h** :

```
#define SYS_settickets 22
#define SYS_getprocessesinfo 23
#define SYS_gettickets 24
```

The code for the system call was added in the file **sysproc.c** :

```

int sys_getprocessesinfo(void) {
    struct processes_info *p;
    if(argptr(0, (void*)&p, sizeof(*p)) < 0)
        return -1;
    acquire(&ptable.lock);
    p->num_processes = 0;
    for(struct proc *pr = ptable.proc; pr < &ptable.proc[NPROC]; pr++) {
        if(pr->state != UNUSED) {
            p->pids[p->num_processes] = pr->pid;
            p->ticks[p->num_processes] = pr->ticks;
            p->tickets[p->num_processes] = pr->tickets;
            p->num_processes++;
        }
    }
    release(&ptable.lock);
    return 0;
}

```

Argument Handling: It first checks if it can access a user-supplied structure (`processes_info`) that will hold the process information. If it fails, it returns an error.

Locking the Process Table: The function acquires a lock on the process table to ensure that no other processes modify it while it is being read, preventing race conditions.

Gathering Process Information: It initializes a counter for the number of processes and iterates through the process table. For each active process, it collects information such as the process ID, CPU ticks used, and scheduling tickets, storing this data in the user-supplied structure.

Releasing the Lock: After collecting the data, it releases the lock on the process table.

Return Statement: Finally, it returns a success code, indicating that the process information has been successfully retrieved.

The following declaration was added in `user.h` :

```
int getprocessesinfo(struct processes_info*);
```

The following line was added in `usys.S` :

```
SYSCALL(getprocessesinfo)
```

Adding support for lottery scheduler :

```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

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

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);

        int total_tickets = 0;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if(p->state == RUNNABLE)
                total_tickets += p->tickets;
        }

        if (total_tickets!=0) {
            int winning_ticket = next_random() % total_tickets;
            for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
                if(p->state == RUNNABLE) {
                    if(winning_ticket < p->tickets) {
                        // Schedule process p
                        c->proc = p;
                        switchvm(p);
                        p->ticks++;
                        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;
                        break;
                    }
                    winning_ticket -= p->tickets;
                }
            }
        }

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

Initialize Total Tickets:

- The variable `total_tickets` is initialized to zero. This variable will accumulate the total number of tickets from all runnable processes.

Accumulate Tickets from Runnable Processes:

- The code iterates over the process table (`ptable.proc`), checking the state of each process. For each process that is in the `RUNNABLE` state (ready to run), its ticket count is added to `total_tickets`. This loop ensures that only processes that can be scheduled contribute to the total.

Check if There Are Any Tickets:

- After summing the tickets, the code checks if `total_tickets` is not zero. If it is zero, it means there are no runnable processes, and the scheduler does not need to proceed further.

Select a Winning Ticket:

- If there are tickets, the code generates a random number, `winning_ticket`, by calling `next_random()` and taking the modulus with `total_tickets`. This gives a random ticket value that falls within the range of total tickets.

Schedule the Winning Process:

- The code again iterates over the process table. For each runnable process, it checks if the current `winning_ticket` is less than the number of tickets for that process.
 - If it is, this means the process has "won" the lottery. The code then:
 - Sets the current CPU's process (`c->proc`) to the winning process (`p`).
 - Calls `switchvm(p)` to switch to the user space of the winning process.
 - Increments the tick count (`p->ticks`) for the winning process to record that it is now running.
 - Changes the process's state to `RUNNING`.
 - Calls `swtch(&(c->scheduler), p->context)` to perform a context switch to the winning process, effectively handing over the CPU to it.
 - Calls `switchkvm()` to switch back to kernel space after the context switch.
 - After the winning process is scheduled, the loop breaks.

Resetting the Current Process:

- Finally, after the scheduled process completes its execution (or is preempted), the current process pointer (`c->proc`) is reset to zero, indicating that there is no currently running process in the context of the CPU.

The file `randomnumber.c` was created to implement the Park - Miller pseudo-random number generator.

```
static unsigned random_seed = 1;

#define RANDOM_MAX ((1u << 31u) - 1u)
unsigned lcg_parkmiller(unsigned *state)
{
    const unsigned N = 0x7fffffff;
    const unsigned G = 48271u;

    /*
     Indirectly compute state*G%N.

     Let:
     div = state/(N/G)
     rem = state%(N/G)

     Then:
     rem + div*(N/G) == state
     rem*G + div*(N/G)*G == state*G

     Now:
     div*(N/G)*G == div*(N - N%G) == -div*(N%G) (mod N)

     Therefore:
     rem*G - div*(N%G) == state*G (mod N)

     Add N if necessary so that the result is between 1 and N-1.
    */
    unsigned div = *state / (N / G); /* max : 2,147,483,646 / 44,488 = 48,271 */
    unsigned rem = *state % (N / G); /* max : 2,147,483,646 % 44,488 = 44,487 */

    unsigned a = rem * G; /* max : 44,487 * 48,271 = 2,147,431,977 */
    unsigned b = div * (N % G); /* max : 48,271 * 3,399 = 164,073,129 */

    return *state = (a > b) ? (a - b) : (a + (N - b));
}

unsigned next_random() {
    return lcg_parkmiller(&random_seed);
}
```

Random Seed:

- A static variable, `random_seed`, is initialized to 1. This seed is used to generate a sequence of pseudo-random numbers. The randomness of the sequence is dependent on the initial seed.

Constants:

- The generator uses two constants:
 - **N**: This represents the maximum value for generated random numbers, set to 2,147,483,647 (which is $2^{31} - 1$).
 - **G**: A multiplier constant set to 48,271. This value is specifically chosen to ensure a good distribution of random numbers.

Linear Congruential Generator (LCG):

- The algorithm works by updating the state (the seed) using a mathematical formula that combines multiplication and modulus operations. The purpose is to generate a new random number based on the current state.

Computation Steps:

- The LCG computes the next state by dividing the current state by a derived value and calculating the remainder. This allows it to break down the state into two components:
 - **div**: This represents the quotient when the current state is divided by N/G .
 - **rem**: This is the remainder of the same division.
- The algorithm then uses these components to derive two intermediate values, `aaa` and `bbb`, which are used to compute the next random number.
- The final random number is determined by ensuring that the result falls within the range of 1 to $N-1$.

Generating Random Numbers:

- The `next_random` function calls the LCG with the current seed and returns a new pseudo-random number, simultaneously updating the seed for the next call.

RUNNING THE TEST PROGRAMS :

The c files were added to **Makefile** :


```
UPROGS=\
_cat\
_echo\
_forktest\
_grep\
_init\
_kill\
_ln\
_ls\
_mkdir\
_rm\
_sh\
_stressfs\
_usertests\
_wc\
_zombie\
_processlist\
_timewithtickets\
_lotterytest\
```

```
EXTRA=\
mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
printf.c umalloc.c processlist.c timewithtickets.c lotterytest.c\
README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
.gdbinit.tmpl gdbutil\
```

Additionally, a system call for the `yield()` function was created to make it accessible by these user level programs.

Processlist.c :

1. Process Information Structure:

- The program begins by declaring a variable `info` of type `processes_info`, which is a structure presumably defined in the system to hold information about processes, such as their process IDs, ticket counts, and tick counts.

2. Retrieving Process Information:

- The program calls the system call `getprocessesinfo`, passing the address of the `info` variable. This system call fills the `info` structure with details about all currently running processes.

3. Displaying the Number of Running Processes:

- After retrieving the process information, the program prints the number of running processes by accessing the `num_processes` field from the `info` structure.

4. Printing Process Headers:

- It prints a header line to label the columns for the subsequent process information output. The headers are for process IDs (PID), the number of tickets assigned to each process, and the number of ticks (CPU time used).

5. Loop Through Processes:

- The program then enters a loop that iterates through the list of processes stored in the `info` structure. For each running process, it prints:
 - The process ID (PID).
 - The number of tickets assigned to that process.
 - The number of ticks that process has accumulated.

6. Exiting the Program:

- Finally, the program calls `exit()` to terminate its execution cleanly.

```
overlapped passed 92.01.92
$ processlist
3 running processes
PID    TICKETS TICKS
1       10     23
2       10     26
66      10     7
$
```

```
processlist
3 running processes
PID    TICKETS TICKS
1       10     23
2       10     28
67      10     1
$ processlist
3 running processes
PID    TICKETS TICKS
1       10     23
2       10     30
68      10     2
$
```

timewithtickets.c:

This code is a C program designed to demonstrate a ticket-based scheduling system in an operating system, specifically within the context of a process management framework similar to xv6.

1. **Functionality:** The program creates multiple child processes, assigns a specified number of "tickets" (which represent CPU time or priority) to each, and monitors their execution for a given number of seconds.

2. **Process Creation:** The `spawn` function forks new processes, sets their ticket count, and either yields (if `USE_YIELD` is defined) or runs indefinitely. Each child process runs a loop that effectively simulates continuous execution.
3. **Ticket Management:** The program uses the `settickets` function to allocate a large number of tickets to itself to avoid being starved by child processes. Each child receives tickets as specified by the command-line arguments.
4. **Waiting for Setup:** The `wait_for_ticket_counts` function yields up to a predefined number of times to ensure that all child processes have their ticket counts set correctly before proceeding.
5. **Execution Duration:** The parent process sleeps for a specified number of seconds to allow the children to run, after which it kills all the child processes.
6. **Process Information Retrieval:** The program uses `getprocessesinfo` to retrieve information about the processes both before and after the sleep period, allowing it to check the status of the child processes, including their ticket counts and execution ticks.
7. **Output:** Finally, the program outputs the number of tickets each child had and the number of ticks (execution time units) they consumed during the specified duration. It also checks for inconsistencies in the process information retrieved.

Overall, this code demonstrates basic process management and scheduling concepts, focusing on the use of tickets to control CPU time allocation among processes in a simulated environment.

```
$ timewithtickets 50 1 10 60
TICKETS TICKS
1         0
10        9
60       41
```

Running the program with 50 as the running time and starting 3 processes with ticket counts of 1, 10 and 60 shows that the process with 60 tickets is running for the largest amount of ticks and the process with 1 tickets is running for the lowest amount of ticks.

```
$ timewithtickets 100 20 30 40
TICKETS TICKS
20       28
30       23
40       49
```

Similarly, processes are being allotted runtime proportional to their ticket count.

lotterytest.c:

Structure of the Code

1. Test Case Definition:

- The `test_case` struct encapsulates the test scenario, including the name, expected and actual ticks, and functions to run for the child processes.
- The test cases cover different scenarios such as unequal ticket distribution, I/O waits, and exit conditions.

2. Process Functions:

- Various functions are defined that represent the behavior of processes:
 - `yield_forever()`: Continuously yields to allow other processes to run.
 - `run_forever()`: Runs indefinitely without yielding.
 - `iowait_forever()`: Simulates an I/O wait condition.
 - `exit_fast()`: Exits immediately.

3. Test Execution:

- `spawn()` is used to create new processes and assign tickets to them.
- `wait_for_ticket_counts()` ensures that processes set their tickets correctly before proceeding.
- `execute_and_get_info()` and `count_ticks()` gather timing information about how many ticks each process received.

4. Statistical Analysis:

- Functions like `compare_ticks_chi_squared()` and `compare_ticks_naive()` perform statistical tests to validate that the ticks received by processes align with their ticket distribution.

5. Debugging and Reporting:

- The framework reports errors when expectations are not met and provides output detailing each test's results.

```
init: starting sh
$ lotterytest
one process: passed 3 of 3
two processes, unequal ratio: passed 7 of 7
two processes, unequal ratio, small ticket count: passed 7 of 7
two processes, equal: passed 7 of 7
two processes, equal, small ticket count: passed 7 of 7
three processes, unequal: passed 9 of 9
three processes, unequal, small ticket count: passed 9 of 9
three processes, but one io-wait: passed 9 of 9
three processes, but one exits: passed 9 of 9
seven processes: passed 17 of 17
two processes, not all yielding: passed 7 of 7
overall: passed 91 of 91
$
```

Running the testing script shows that all test cases have passed.

PART B:

Creation of system calls **getNumProc()** and **getMaxPID()**:

User programs `getNumProcTest` and `getMaxPIDTest` are created to call these system calls. Relevant functions `getNumProcAssist` and `getMaxPIDAssist` are added in `proc.c`.

```

int getNumProcAssist(void){
    int ans=0;
    struct proc *p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != UNUSED)
            ans++;
    }
    release(&ptable.lock);

    return ans;
}

int getMaxPIDAssist(void){
    int max=0;
    struct proc *p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != UNUSED){
            if(p->pid > max)
                max=p->pid;
        }
    }
    release(&ptable.lock);

    return max;
}

```

As we can notice, the functions acquire lock on the process table (ptable) and then iterates through the loop to carry out respective tasks.

To see this in action, run the following commands:

```
ls
getMaxPIDTest
getNumProcTest
```

One after the other.

Output:

```
udbhav@Udbhav514: ~/xv6-public (another copy)
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat       2 3 19348
echo      2 4 18224
forktest  2 5 9072
grep      2 6 22184
init      2 7 18800
kill      2 8 18312
ln        2 9 18212
ls        2 10 20780
ioProcTester 2 11 20288
hybridtester 2 12 18464
sh        2 13 32324
stressfs  2 14 19244
usertests 2 15 66664
cpuProcTester 2 16 20492
zombie    2 17 17888
getNumProcTest 2 18 18000
getMaxPIDTest 2 19 17992
getProcInfoTest 2 20 18736
burstTimeTest 2 21 18156
test_scheduler 2 22 21260
console   3 23 0
$
```

```
$ getMaxPIDTest
The maximum PID of all active processes in the system is 4
$
```

```
$ getNumProcTest
The total number of active processes in the system is 3
$
```

PIDs 1 and 2 are allocated to system processes, hence when `ls` is called, it is assigned PID 3. Now when `getMaxPIDTest` is called, `ls` will have terminated, and it will be assigned PID 4, which is the maximum till now, hence returning 4, after which `getMaxPIDTest` terminates. Now when `getNumProcTest` is called, there are only 3 active processes - 2 system processes and 1 `getNumProcTest`, hence the output is 3.

To implement the `getProcInfo` system call, we need to pass arguments using `argptr`, which is designed for this purpose. Additionally, to track context switches, we modify `struct proc` by adding a `nocs` field.

We initialize `nocs` to 0 in `allocproc()`, as it's called before a process runs. Then, in `scheduler()`, we increment `p->nocs` to track context switches.

A dummy process, `defaultParent`, is created, and we assign it as the parent of every process using `p->parent = &defaultParent` in `allocproc()`. `fork()` later replaces it with the actual parent. The `defaultParent` PID is set to -2, allowing us to check if a process has a parent and retrieve its PID using `p->parent->pid`.

```
int
sys_getProcInfo(void){
    int pid;

    struct processInfo *info;
    argptr(0,(void *)&pid, sizeof(pid));
    argptr(1,(void *)&info, sizeof(info));

    struct processInfo temporaryInfo = getProcInfoAssist(pid);

    if(temporaryInfo.ppid == -1)return -1;

    info->ppid = temporaryInfo.ppid;
    info->psize = temporaryInfo.psize;
    info->numberContextSwitches = temporaryInfo.numberContextSwitches;
    return 0;
}
```



```

struct processInfo getProcInfoAssist(int pid){

    struct proc *p;
    struct processInfo temp = {-1,0,0};

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != UNUSED){
            // printf(1, "%d\n", p->pid);
            if(p->pid == pid) {
                temp.ppid = p->parent->pid;
                temp.psize = p->sz;
                temp.numberContextSwitches = p->nocs;
                release(&ptable.lock);
                return temp;
            }
        }
    }
    release(&ptable.lock);

    return temp;
}

```

We use a dummy variable `temp` of type `processInfo` to indicate when no process exists with the given PID. In `getProcInfoTest`, the process ID is passed as a command-line parameter to `getProcInfo`, which forwards it to `getProcInfoAssist`. This function iterates over the process table (`ptable`) to gather the required information, returning it in a `processInfo` variable, from which the relevant data is extracted.

```

$ getProcInfoTest 1
PPID: No Parent Process
Size: 16384
Number of Context Switches: 24
$ getProcInfoTest 5
exec: fail
exec getProcInfoTest failed
$ getProcInfoTest 3
No process has that PID.
$ getProcInfoTest 2
PPID: 1
Size: 20480
Number of Context Switches: 34
$

```

To track burst time, we add a `burst_time` attribute to the `proc` structure. We implement two system calls, `set_burst_time` and `get_burst_time`, to set and retrieve the burst time of the current process. The default value of `burst_time` is set to 0 in `allocproc()`.

To access the current process, we use the `myproc()` function, which returns a pointer to the `proc` structure of the current process. This allows easy modification of the burst time.

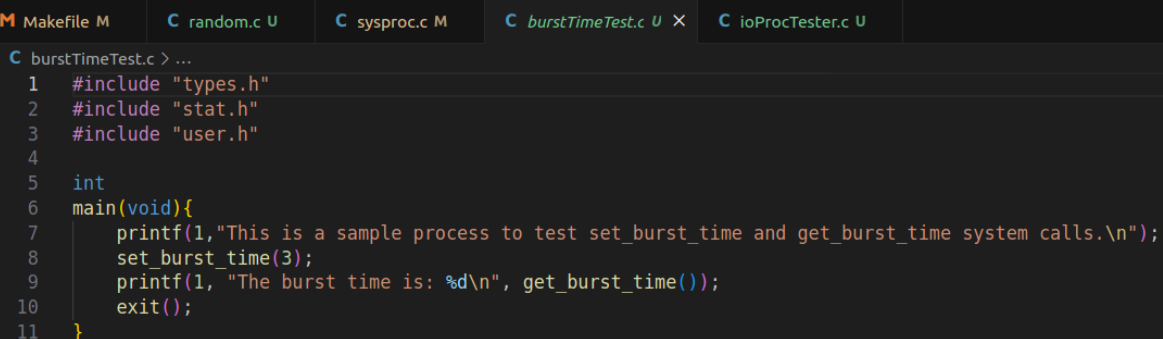
```
int
set_burst_timeAssist(int burst_time)
{
    struct proc *p = myproc();
    p->burst_time = burst_time;
    if(burst_time < quant){
        quant = burst_time;
    }
    yield();

    return 0;
}

int
get_burst_timeAssist()
{
    struct proc *p = myproc();

    return p->burst_time;
}
```

User program `burst_time_test` is create to test relevant processes.



```
M Makefile M    C random.c U    C sysproc.c M    C burstTimeTest.c U X    C ioProcTester.c U
C burstTimeTest.c > ...
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int
6  main(void){
7      printf(1,"This is a sample process to test set_burst_time and get_burst_time system calls.\n");
8      set_burst_time(3);
9      printf(1, "The burst time is: %d\n", get_burst_time());
10     exit();
11 }
```

```
$ burstTimeTest
This is a sample process to test set_burst_time and get_burst_time system calls.
The burst time is: 3
$
```

It can be noticed that burst time is set to 3. It changes from 0 to 3 indicating that the above functions are working correctly.

PART C:

For this part, we shall implement Shortest Job First (SJF) instead of Round Robin Scheduling as in part B.

To prevent the current process from being preempted on every OS clock tick, allowing it to run until completion, we modify the round-robin scheduler. The forced preemption occurring on each clock tick is handled in `trap.c`. To resolve this, simply remove the following lines from `trap.c`:

```
if(myproc() && myproc()->state == RUNNING &&
    tf->trapno == T_IRQ0+IRQ_TIMER)
```

To modify the scheduler so that processes are executed in increasing order of their burst times, we implemented a priority queue (min heap) using an array, which sorts processes by burst time. The heap is locked for safety. At any given time, the queue contains all `RUNNABLE` processes. When the scheduler selects the next process, it calls `extract_min` to pick the process with the shortest burst time from the front of the queue. The required modifications were made in `proc.c`.

Declaration of Priority Queue:

```
struct {
    struct spinlock lock;
    int siz;
    struct proc* proc[NPROC+1];
} pqueue;
```

Implementation of relevant functions:

- insertIntoHeap (Inserts a given process into the priority queue):

```

void insertIntoHeap(struct proc *p){
    if(isFull())
        return;

    acquire(&pqueue.lock);

    pqueue.siz++;
    pqueue.proc[pqueue.siz]=p;
    int curr=pqueue.siz;
    while(curr>1 && ((pqueue.proc[curr]->burst_time)<(pqueue.proc[curr/2]->burst_time))){
        struct proc* temp=pqueue.proc[curr];
        pqueue.proc[curr]=pqueue.proc[curr/2];
        pqueue.proc[curr/2]=temp;
        curr/=2;
    }
    release(&pqueue.lock);
}

```

- isEmpty (Checks if the priority queue is empty or not):

```

int isEmpty(){
    acquire(&pqueue.lock);
    if(pqueue.siz == 0){
        release(&pqueue.lock);
        return 1;
    }
    else{
        release(&pqueue.lock);
        return 0;
    }
}

```

- isFull (Checks if the priority queue is full or not):

```

int isFull(){
    acquire(&pqueue.lock);
    if(pqueue.siz==NPROC){
        release(&pqueue.lock);
        return 1;
    }
    else{
        release(&pqueue.lock);
        return 0;
    }
}

```

- extractMin (removes the process at the front of the queue and returns it):

```

struct proc * extractMin(){
    if(isEmpty())
        return 0;

    acquire(&pqueue.lock);
    struct proc* min=pqueue.proc[1];
    if(pqueue.siz==1)
    {
        pqueue.siz=0;
        release(&pqueue.lock);
    }
    else{
        pqueue.proc[1] = pqueue.proc[pqueue.siz];
        pqueue.siz--;
        release(&pqueue.lock);

        fix(1);
    }
    return min;
}

```

- changeKey (Changes the burst time of a process with a given PID in the priority queue and updates the queue accordingly):

```

void changeKey(int pid, int x){

    acquire(&pqueue.lock);

    struct proc* p;
    int curr=-1;
    for(int i=1;i<=pqueue.siz;i++){
        if(pqueue.proc[i]->pid == pid){
            p=pqueue.proc[i];
            curr=i;
            break;
        }
    }

    if(curr==-1){
        release(&pqueue.lock);
        return;
    }

    if(curr==pqueue.siz){
        pqueue.siz--;
        release(&pqueue.lock);
    }
    else{
        pqueue.proc[curr]=pqueue.proc[pqueue.siz];
        pqueue.siz--;
        release(&pqueue.lock);

        fix(curr);
    }

    p->burst_time=x;
    insertIntoHeap(p);
}

```

- fix (performs Heapify on priority queue - basically converts the array into min heap assuming that the left subtree and the right subtree of the root are already min heaps):

```

void fix(int curr){
    acquire(&pqueue.lock);
    while(curr*2<=pqueue.siz){
        if(curr*2+1<=pqueue.siz){
            if((pqueue.proc[curr]->burst_time)<=(pqueue.proc[curr*2]->burst_time)&&(pqueue.proc[curr]->burst_time)<=(pqueue.proc[curr*2+1]->burst_time))
                break;
            else{
                if((pqueue.proc[curr*2]->burst_time)<=(pqueue.proc[curr*2+1]->burst_time)){
                    struct proc* temp=pqueue.proc[curr*2];
                    pqueue.proc[curr*2]=pqueue.proc[curr];
                    pqueue.proc[curr]=temp;
                    curr*=2;
                } else {
                    struct proc *temp
                    struct proc* temp=pqueue.proc[curr*2+1];
                    pqueue.proc[curr*2+1]=pqueue.proc[curr];
                    pqueue.proc[curr]=temp;
                    curr*=2;
                    curr++;
                }
            }
        } else {
            if((pqueue.proc[curr]->burst_time)<=(pqueue.proc[curr*2]->burst_time))
                break;
            else{
                struct proc* temp=pqueue.proc[curr*2];
                pqueue.proc[curr*2]=pqueue.proc[curr];
                pqueue.proc[curr]=temp;
                curr*=2;
            }
        }
    }
    release(&pqueue.lock);
}

```

Change the scheduler so that it uses the priority queue to schedule the next process.
Priority queue functions acquire and release priority queue locks.

```

void
scheduler(void)
{
    defaultParent.pid = -2;
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

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

        acquire(&ptable.lock);

        if(isEmpty()){
            if(isEmpty2()){
                goto label;
            }

            while(!isEmpty2()){
                if((p = extractMin2()) == 0){release(&ptable.lock);break;}
                insertIntoHeap(p);
            }
        }

        label:

        if((p = extractMin()) == 0){release(&ptable.lock);continue;}

        if(p->state!=RUNNABLE)
            {release(&ptable.lock);continue;}

        c->proc = p;
        switchvm(p);

        p->state = RUNNING;
        (p->nocs)++;

        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;

        release(&ptable.lock);
    }
}

```

Processes are inserted into the priority queue whenever their state changes to **RUNNABLE**. This occurs in five functions: **yield**, **kill**, **fork**, **userinit**, and **wakeup1**. Below is the code for

the `fork` function, with the others being similar. A variable `check` is used to verify if the process is already in the `RUNNABLE` state, ensuring it is not reinserted into the priority queue.

```
acquire(&ptable.lock);

short check = (np->state!=RUNNABLE);
np->state = RUNNABLE;

//Insert Process Into Queue
if(check)
    insertIntoHeap(np);

release(&ptable.lock);

return pid;
```

Add a `yield` call within the `set_burst_time` function. This ensures that when the burst time of a process is updated, the scheduler can make decisions based on the new burst time. The `yield` function changes the state of the current process to `RUNNABLE`, inserts it into the priority queue, and switches context to the scheduler.

```
int
set_burst_timeAssist(int burst_time)
{
    struct proc *p = myproc();
    p->burst_time = burst_time;
    if(burst_time < quant){
        quant = burst_time;
    }
    yield();

    return 0;
}
```

Runtime Complexity:

The runtime complexity of the scheduler is dominated by the `extractMin` operation, which has a time complexity of $O(\log n)$. The other statements in the scheduler function run in constant time.

Corner Case Handling and Safety:

- **Empty Queue:** If the queue is empty, `extractMin` returns 0, preventing the scheduler from scheduling any processes.
- **Full Queue:** If the priority queue is full, `insertIntoHeap` rejects the new process and returns without removing any existing processes.
- **Avoiding Duplicates:** When inserting a process into the priority queue, the scheduler checks if the process was already in the queue by verifying its state before it becomes `RUNNABLE`. If it was already `RUNNABLE`, it won't be reinserted.
- **Robust Functions:** The priority queue functions are designed to be robust, minimizing the risk of segmentation faults.
- **State Checking:** The scheduler verifies that the process at the front of the queue is `RUNNABLE`. If it is not, the scheduler skips scheduling that process, protecting the system from potential issues.
- **Zombie Process Handling:** When freeing `ZOMBIE` child processes, the scheduler checks the priority queue and removes any related processes using `changeKey` and `extractMin`.
- **Data Consistency:** To maintain data consistency, a dedicated lock is used when accessing the priority queue. This lock is created specifically for the priority queue and is initialized in the `pinit` function.

Testing

We conducted thorough testing to ensure that our new scheduler is robust and functions correctly under various conditions. This involved forking multiple processes with different burst times, with approximately half being CPU-bound processes and the other half I/O-bound processes.

- **CPU-Bound Processes:** These processes consist of loops that run for many iterations (up to 10810^8). We discovered that compilers often optimize away loops if the computed values are not used afterward. To counter this, we ensured that the results of computations within the loops were utilized.
- **I/O-Bound Processes:** We simulated I/O-bound processes by calling `sleep(1)` 100 times. The `sleep` function changes the state of the current process to sleeping for a specified number of clock ticks, which mimics the behavior of processes waiting for user input. When one I/O-bound process sleeps, the scheduler switches the context to another runnable process.

To verify that the Shortest Job First (SJF) scheduler operates based on burst times, we developed a program called `test_scheduler`. This program takes an argument specifying the number of forked processes and returns statistics for each executed process.

```

void
pinit(void)
{
    initlock(&ptable.lock, "ptable");
    initlock(&pqueue.lock, "pqueue");
    initlock(&pqueue2.lock, "pqueue2");
}

```

All CPU-bound and I/O-bound processes are successfully sorted by their burst times, with CPU-bound processes completing first. This occurs because the I/O-bound processes are blocked by the sleep system call. The sorted execution confirms that the Shortest Job First (SJF) scheduler is functioning correctly.

Context switches occur as expected: for CPU-bound processes, once a process is scheduled, `set_burst_time` is called, leading to a yield and the scheduling of the next process. When the previously yielded process is rescheduled, it completes its execution. I/O-bound processes are also put to sleep 100 times, resulting in an additional 100 context switches as they are reintroduced into the scheduler.

To compare the performance of the SJF scheduler with the default Round Robin scheduler, we developed two test programs: `cpuProcTester`, which runs CPU-bound processes, and `ioProcTester`, which exclusively runs I/O-bound processes. This setup simplifies the comparison between the two scheduling algorithms.

Outputs of Round Robin Scheduler:

```

$ cpuProcTester 10
  PID    Type    Burst Time    Context Switches
  ----    -
  31      CPU      252            10
  28      CPU      264            10
  34      CPU      270            10
  25      CPU      635            10
  33      CPU      822            10
  30      CPU      985            10
  26      CPU      999            10
$ ioProcTester 10
  PID    Type    Burst Time    Context Switches
  ----    -
  38      I/O      21            12
  40      I/O      71            12
  43      I/O      126           12
  42      I/O      252           12
  39      I/O      264           12
  45      I/O      270           12
  36      I/O      635           12
  44      I/O      822           12
  41      I/O      985           12
  37      I/O      999           12

```

Outputs of SJF Scheduler:

In the Round Robin (RR) scheduler, the execution order is closely tied to the process ID (PID), as it uses a First-Come, First-Served (FCFS) queue. In contrast, the Shortest Job First (SJF) scheduler organizes processes based on their burst times, leading to more efficient scheduling. Additionally, the number of context switches in the RR scheduler is significantly higher due to forced preemption at every clock tick.

```
$ cpuProcTester 10
```

PID	Type	Burst Time	Context Switches
4	CPU	13	2
5	CPU	20	2
6	CPU	1	2
7	CPU	6	2
8	CPU	2	2
9	CPU	20	2
10	CPU	6	2
11	CPU	3	2
12	CPU	17	2
13	CPU	6	2

```
$ ioProcTester 10
```

PID	Type	Burst Time	Context Switches
15	I/O	13	22
16	I/O	20	22
17	I/O	1	22
18	I/O	6	22
19	I/O	2	22
20	I/O	20	22
21	I/O	6	22
22	I/O	3	22
23	I/O	17	22
24	I/O	6	22

```
$
```

Testing Notes:

- Burst times are generated using a random number generator implemented in the `random.c` file, which has been made into a user library.
- We removed `wc` and `mkdir` from the Makefile, as we could not have more than 20 user programs in `UPROGS`. The operating system encountered compilation issues with a larger number of user programs due to virtual hard drive limitations.
- The `set_burst_time` function yields the current process, ensuring that the new burst times are considered during scheduling.

PART C (BONUS):

We implemented a hybrid scheduler that combines elements of Round Robin and Shortest Job First (SJF) scheduling by modifying the `trap.c`, `proc.c`, and `defs.h` files.

Time Quantum Setup:

1. **Variable Declaration:** We declared an external integer variable `quant` in `defs.h` to be initialized in `proc.c`. It is set to a default value of 1000, as the burst times range from 1 to 1000.
2. **Burst Time Adjustment:** The assignment requires the time quantum to match the burst time of the process with the shortest burst time in the priority queue. Since the default burst time is zero, we check if the burst time to be set is less than `quant`. If so, we update `quant` to the new burst time in the `set_burst_time` function.

Priority Queue Implementation:

- We created a second priority queue, `pqueue2`, along with corresponding functions. Functions like `insertIntoHeap`, `fix`, and `extractMax` were replicated for `pqueue2` with corresponding names (e.g., `insertIntoHeap2`, `extractMax2`, `fix2`). These functions perform the same operations as their counterparts but operate on `pqueue2`.

Clock Tick Interrupt Handler Modification:

- In `trap.c`, we modified the clock tick interrupt handler. At each clock tick, we increment the running time (`rt`) of the current process (accessed via `myproc()`), which is initialized to zero when the process is created in `allocproc`.
- Since processes default to a burst time of zero, we avoid terminating them immediately after their first clock tick. We only check for equivalence between `rt` and `burst_time` if `burst_time` is non-zero.
- If the running time matches the burst time, the current process is exited. If not, we check if `rt` is divisible by the time quantum `quant`. If true, we preempt the current process and insert it into `pqueue2`.

```

void
scheduler(void)
{
    defaultParent.pid = -2;
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){
        sti();
        acquire(&ptable.lock);

        if(isEmpty()){
            if(isEmpty2()){
                goto label;
            }

            while(!isEmpty2()){
                if((p = extractMin2()) == 0){release(&ptable.lock);break;}
                insertIntoHeap(p);
            }
        }

        label:
        if((p = extractMin()) == 0){release(&ptable.lock);continue;}

        if(p->state!=RUNNABLE)
            {release(&ptable.lock);continue;}

        c->proc = p;
        switchvm(p);

        p->state = RUNNING;
        (p->nocs)++;

        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;
        release(&ptable.lock);
    }
}

```

```

if (myproc() && myproc()->state == RUNNING &&
    tf->trapno == T_IRQ0 + IRQ_TIMER)
{
    (myproc()->rt)++;
    if (myproc()->burst_time != 0)
    {
        if (myproc()->burst_time == myproc()->rt)
            exit();
    }
    if ((myproc()->rt) % quant == 0)
        new_yield();
}

void new_yield(void){
    acquire(&ptable.lock);

    myproc()->state = RUNNABLE;

    insertIntoHeap2(myproc());

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

```

Next, we modified the scheduler to enhance its functionality. The updated scheduler operates as follows:

1. **Process Selection:** When determining which process to run next, the scheduler first checks if the original priority queue, `pqueue`, is empty.
2. **Extracting from `pqueue`:**
 - If `pqueue` is not empty, the scheduler calls `extractMin` to retrieve and run the process at the front of the queue.
3. **Handling an Empty `pqueue`:**
 - If `pqueue` is empty, the scheduler removes all processes from `pqueue2` (those that were preempted due to the time quantum) and inserts them back into `pqueue`.
 - After reinserting the preempted processes, the scheduler then selects the next process from `pqueue` using `extractMin` as usual.

This approach allows the scheduler to effectively manage both CPU-bound and I/O-bound processes, ensuring that processes that were preempted due to time quantum can be resumed in an orderly manner.

The final part of our implementation involved testing using three user programs: ``test_scheduler``, ``cpuProcTester``, and ``ioProcTester``. Here's a summary of what was done in each program:

Testing Programs:

1. ``test_scheduler.c``:
 - Half of the forked processes are I/O-bound, and the other half are CPU-bound.
 - Each process is assigned a random burst time between 1 and 1000.
 - CPU-bound processes run loops for $\backslash(10^9\backslash)$ iterations.
 - I/O-bound processes call ``sleep(1)`` ten times to simulate I/O operations.

2. `cpuProcTester.c`:

- All forked processes are CPU-bound.
- Burst times are randomly assigned between 1 and 1000.
- Each process runs a loop of (10^9) iterations, which takes approximately 200 ticks to complete.

3. `ioProcTester.c`:

- Every forked process is I/O-bound.
- I/O simulation is performed by calling `sleep(1)` ten times.
- Burst times are also randomly assigned with values between 1 and 1000.

Output:

The outputs obtained from these tests effectively demonstrate the functionality and performance of the new hybrid scheduler. They validate that CPU-bound processes and I/O-bound processes are being managed as expected, with the SJF and Round Robin scheduling features working in harmony.


```
$ cpuProcTester 10
```

PID	Type	Burst Time	Context Switches
----	----	-----	-----
31	CPU	252	10
28	CPU	264	10
34	CPU	270	10
25	CPU	635	10
33	CPU	822	10
30	CPU	985	10
26	CPU	999	10

```
$ ./ioProcTester 20 -fitted
```

```
$ ioProcTester 20
```

PID	Type	Burst Time	Context Switches
----	----	-----	-----
70	I/O	21	12
85	I/O	67	12
72	I/O	71	12
75	I/O	126	12
74	I/O	252	12
71	I/O	264	12
77	I/O	270	12
84	I/O	411	12
81	I/O	443	12
68	I/O	635	12
79	I/O	675	12
83	I/O	677	12
86	I/O	738	12
76	I/O	822	12
87	I/O	846	12
82	I/O	889	12
80	I/O	932	12
78	I/O	962	12
73	I/O	985	12
69	I/O	999	12

```
$ █
```

```
$ test_scheduler 20
```

PID	Type	Burst Time	Context Switches
----	----	-----	-----
91	I/O	21	12
93	I/O	71	12
95	I/O	252	12
105	I/O	411	12
89	I/O	635	12
107	I/O	738	12
97	I/O	822	12
103	I/O	889	12
101	I/O	932	12
99	I/O	962	12
92	CPU	264	10
98	CPU	270	10
102	CPU	443	10
100	CPU	675	10
104	CPU	677	10
108	CPU	846	10
94	CPU	985	10
90	CPU	999	10

```
$
```

Results

Here are the key observations from our testing results, which aligned with our expectations:

1. Completion of CPU-Bound Processes:

- Not all CPU-bound processes completed their execution. Some were killed before they printed any output because their burst time was lower than their actual execution time. This demonstrates that when the `rt` (running time) value of a process equals its `burst_time`, the process is exited.

2. Increased Context Switches:

- The new hybrid scheduling resulted in a significantly higher number of context switches for CPU-bound processes. This is due to preemption occurring every `quant` clock ticks, leading to more frequent interruptions.

3. Behavior of I/O-Bound Processes:

- I/O-bound processes were not significantly affected by the preemption and behaved similarly to those in SJF scheduling. Since they spend most of their time sleeping, their actual execution time is minimal. The likelihood of hitting `quant` clock ticks is low because `quant` tends to be a 2-3 digit number, given that burst times are randomly chosen between 1 and 1000. As a result, their context switch count remained consistent with SJF scheduling.

4. Preemption Among CPU-Bound Processes:

- When using a mix of CPU and I/O-bound processes, some CPU-bound processes experienced more preemptions than others. This was caused by I/O-bound processes returning from the SLEEPING state, which forced the currently running CPU-bound processes to yield, leading to out-of-order execution for some CPU-bound processes.

5. Comparison with Round Robin Scheduler:

- For those interested in comparing the results with the Round Robin scheduler, refer to the pictures provided in the earlier section (non-BONUS section) for a visual representation of the differences.

Overall, the results validate the effectiveness of our hybrid scheduler in managing both CPU-bound and I/O-bound processes while highlighting the nuances of their execution patterns.