

REPORT

1. Gotta count 'em all :

File changed :

In kernel : sysproc.c , proc.c proc.h ,syscall.c

In user : syscount.c , user.h ,usys.pl

In sysproc.c added 2 functions

- i) `int sys_getSysCount void :`
- ii) `int sum_syscall_counts(struct proc *p int syscall_num)`

1. sum_syscall_counts(struct proc *p, int syscall_num):

Purpose: This function recursively calculates the total number of times a specific system call (syscall_num) has been invoked by a given process (p) and all of its descendant (child) processes.

Retrieves the count of the specified system call (syscall_num) from the current process's syscall_count array and initializes the count variable with this value.

- **Loop Initialization:** Iterates through all possible processes in the system (assuming proc is the base array of all processes and NPROC is the maximum number of processes).
- **Child Identification:** Checks if the current child process has p as its parent (child->parent1 == p).
- **Recursive Summation:** If the child belongs to p, the function calls itself recursively to add the child's count of the specified system call to the total count

2. sys_getSysCount(void)

Purpose: This system call retrieves the total number of invocations of a specific system call across the current process and all its descendant processes. The specific system call to count is identified using a bitmask provided as an argument.

- **Bit Position Identification:** Shifts the mask right until the remaining value is 1, effectively finding the position of the single set bit. This position corresponds to the system call number.
- **Bounds Checking:** Ensures that the derived syscall_num does not exceed the maximum allowed number of system calls (MAX_SYSCALLS). If it does, the function returns -1 to indicate an invalid system call number.
- Calls the previously defined sum_syscall_counts function to calculate the total number of invocations of the specified system call (syscall_num) for the current process and all its descendant processes.

In proc.h : add this in the proc struct to keep track of the parent

```
struct proc *parent1;
int syscall_count[MAX_SYSCALLS]; // Track up to 31 system calls (assuming maximum 31 syscalls)
```

In proc.c :

```
for (int i = 0; i < 31; i++)
{
    p->syscall_count[i] = 0;
}
```

Initialized syscallcount for all 32 commands

In syscall.c :

```
void syscall(void){
    // existing code
    p->syscall_count[num]++; // added this line to increase the syscallcount of the given call
    //
}
```

```
[SYS_getSysCount] sys_getSysCount, // add this in this static uint64 (*syscalls[])(void) = {
}
```

2. Wake me up when my timer ends [13 points]:

FILES Changed :

In kernel : trap.c , sysproc.c, syscall.c .proc.h

In user: user.h ,usys.pl

In proc.h : Added this in proc

```
Struct proc {  
int syscall_count[MAX_SYSCALLS]; // Track up to 31 system calls (assuming maximum 31 syscalls)  
int alarmticks; // How many ticks before the alarm goes off  
int ticks_elapsed; // Ticks elapsed since last alarm  
void (*alarmhandler)();  
}  
// Function to call on alarm
```

In sysproc.c : Added this 2 functions

```
Uint64 sys_sigalarm(void);  
Uint64 sys_sigreturn(void);
```

sys_sigalarm

1. Retrieves two arguments: ticks (after which the alarm triggers) and handler (function to call).
2. Gets the current process (p) using myproc().
3. Locks the process (acquire(&p->lock)), sets the alarm interval, resets tick count, assigns the handler, and clears the handler flag (p->in_handler = 0).
4. Releases the lock and returns 0.

Purpose: This function sets up a timer to trigger a user-defined handler after a certain number of ticks.

sys_sigreturn

1. Retrieves the current process (p) and locks it.
2. Checks if the process is in a handler (p->in_handler). If not, releases the lock and returns -1.
3. Restores the process's trapframe from backup (p->tf_backup) to revert to the previous state.

4. Resets the handler flag (`p->in_handler = 0`), releases the lock, and returns 0.

Purpose: This function allows a process to return from an alarm handler, restoring its previous state and resuming normal execution.

In `trap.c` :

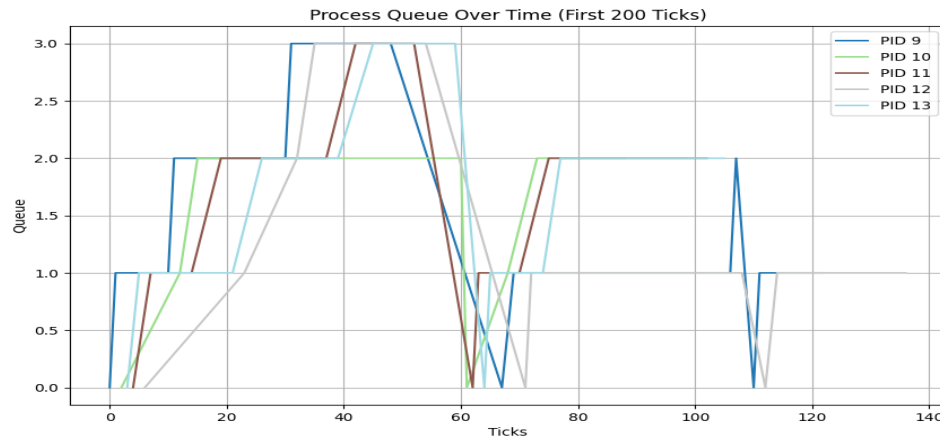
```
void usertrap(void)
{
    // existing code

    if (which_dev == 2)
    {
        // changes are made here
        yield();
    }
    // existing code
}
```

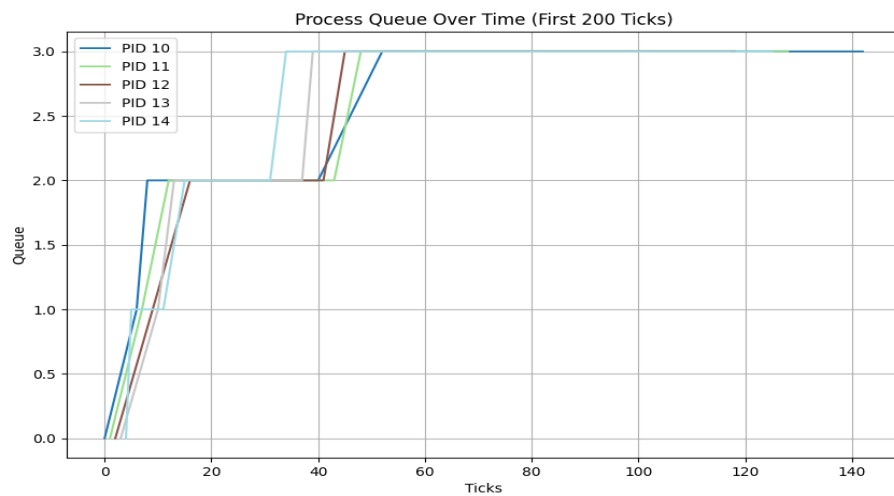
- **Alarm Handling:**
- If the process has an active alarm (`p->alarmticks > 0`), it increments `p->ticks_elapsed`, tracking the number of ticks since the alarm was set.
- **Triggering the Alarm:**
- When `p->ticks_elapsed` equals or exceeds `p->alarmticks`, the alarm is triggered. The handler is invoked only if `p->in_handler` is 0, to prevent reentrancy (re-entering the handler while it's already running).
- **Entering the Alarm Handler:**
- Before invoking the handler, the current trapframe (which stores the process's CPU state) is backed up to `p->tf_backup` for later restoration by `sys_sigreturn`.
- **Setting the Program Counter:**
- The program counter (`p->trapframe->epc`) is set to the alarm handler's address, ensuring that the handler function is executed next time the process is scheduled.
- **Yield:**
- After the above, the `yield()` function is called to potentially switch to another process, enabling cooperative multitasking.

Scheduling :

1. MLFQ : with boosting



Without boosting :



Changes made in files : trap.c , proc.c,proc.h

In proc.h :

```
Struct proc {  
int level; // Current priority level (0 to 3)  
int in_queue; // Flag to check if in a queue  
int enter_ticks; // Ticks when entered the queue  
int ticks;  
}
```

In trap.c

```
void usertrap(void){
p->ticks++;
}
void clockintr()
{
// increased ticks
}
```

In proc.c

```
struct proc *queues[NQUEUE][NPROC]; // Array of queues
int queue_sizes[NQUEUE]; // Sizes of each queue
int current_time = 0;
// Function to push a process to the specified queue
void push(int queue_num, struct proc *p)
struct proc *pop(int queue_num)
void scheduler(void){
// added schedulling logic
}
```

1. **Aging and Cleanup:** It first iterates through all processes, removing any that are in the ZOMBIE state to clean up terminated processes.
2. **Priority Boosting:** Every 48 ticks, it boosts all active processes to the highest priority queue (level 0) to prevent starvation and ensure fairness.
3. **Queue Management:** It ensures that each runnable process is placed in the correct priority queue based on its current level, updating their entry ticks.
4. **Scheduling Execution:** The scheduler then loops through the queues from highest to lowest priority, selecting runnable processes to run. Each process is given a time slice based on its priority level.
5. **Demotion:** If a process uses up its time slice without completing, it is moved to a lower priority queue, allowing higher priority processes to be scheduled first.
6. **Concurrency Control:** Locks are used when accessing and modifying process states to ensure thread safety during scheduling operations.

2.LBS :

Changes made in the files: proc.h , proc.c

In proc.h ;

```
Struct proc {  
int tickets; // Number of lottery tickets  
uint creation_time;  
}
```

In proc.c:

```
int next = 1; // Initial seed for the random number generator  
  
// Linear Congruential Generator (LCG) to generate pseudo-random numbers  
int my_rand();  
  
// Function to generate a random number in the range [0, max]  
int random_at_most(int max)  
  
void scheduler(void){  
// logic here  
}
```

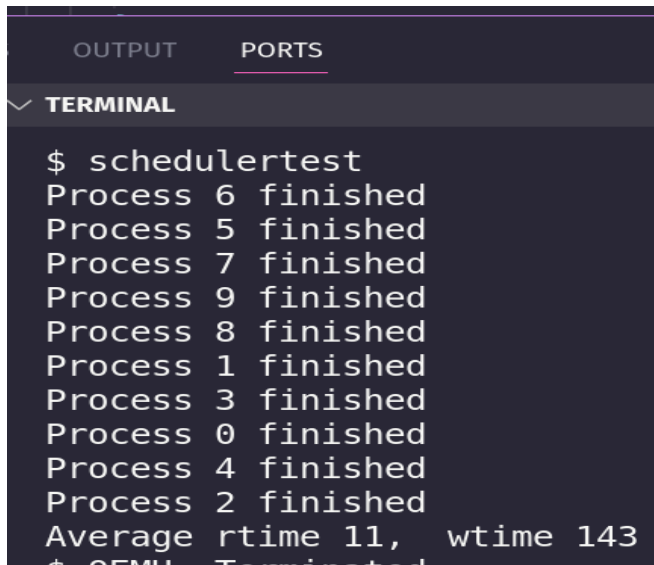
a **Lottery Scheduler**, a probabilistic CPU scheduling algorithm where each runnable process holds a certain number of tickets, and the scheduler randomly selects a ticket to choose the next process to run. Here's a breakdown of the steps:

1. **Total Ticket Calculation:** It first iterates through all processes, acquiring locks to safely sum up the total number of tickets held by all processes in the RUNNABLE state.
2. **Random Ticket Selection:** If there are runnable processes, it generates a random number (winning_ticket) between 0 and the total number of tickets to determine the winning ticket.
3. **Process Selection:** It then iterates through the processes again, accumulating ticket counts until it reaches or exceeds the winning_ticket. If multiple processes have the same number of tickets, it resolves ties by selecting the process that was created earlier.

4. **Context Switching:** Once the winning process is chosen, the scheduler sets its state to RUNNING, updates the current process context (c->proc), and performs a context switch to execute the selected process.
5. **Concurrency Handling:** Throughout the process selection, locks are used to ensure that process states and ticket counts are accessed and modified safely, preventing race conditions.

Comparison between LBS and MLFQ :

i)LBS



```
OUTPUT  PORTS
✓ TERMINAL
$ schedulertest
Process 6 finished
Process 5 finished
Process 7 finished
Process 9 finished
Process 8 finished
Process 1 finished
Process 3 finished
Process 0 finished
Process 4 finished
Process 2 finished
Average rtime 11, wtime 143
$ O5ML: Terminated
```

MLFQ :


```
xv6 kernel is booting

init: starting sh
$ schedulertest
Process 5 finished
Process 9 finished
Process 6 finished
Process 7 finished
Process 8 finished
Process 0 finished
Process 1 finished
Process 2 finished
Process 3 finished
Process 4 finished
Average rtime 12, wtime 140
#
```

What is the implication of adding the arrival time in the lottery based scheduling policy?

A: Arrival Time Fairness: Processes with the same number of tickets are prioritized based on arrival time, ensuring earlier processes are favored when ticket counts are equal.

Are there any pitfalls to watch out for?

A: Pitfall - Starvation of Newer Processes: Newer processes may suffer starvation if older processes with the same number of tickets dominate CPU time, especially when many processes share similar ticket counts.

What happens if all processes have the same number of tickets?

A: Default to First-Come-First-Served: If all processes have the same number of tickets, the system effectively switches to first-come-first-served based on arrival time, ensuring fairness without randomness