COL331 Operating Systems Assignment 2

Abhinav Rajesh Shripad **Entry Number: 2022CS11596**
Jahnabi Roy **Entry Number: 2022CS11094**

April 9, 2025

Signal Handling in xv6

- For all the subsections of this part, changes were done in `consoleintr` function found in `console.c`. `Ctrl-C, Ctrl-B, Ctrl-G and Ctrl-F` keys were configured to be recognized which sets their corresponding bits to 1 and calls the responsible function accordingly.

```c
void consoleintr(int (*getc)(void))
{
    (...)
    int tokill = 0, tofreeze = 0, torestart = 0, custom = 0;

    (...)
    case C('C'):
        tokill = 1;
        break;
    case C('F'):
        torestart = 1;
        break;
    case C('B'):
        tofreeze = 1;
        break;
    case C('G'):
        custom = 1;
        break;
    (...)

    if (tokill)
    {
        ctrlkill();
    }
    if (tofreeze)
    {
        ctrlfreeze();
    }
    if (torestart)
    {
        ctrlrestart();
    }
    if (custom)
    {
        registerctrl();
    }

    (...)
}
```

0.1 SIGINT (Signal Interrupt)

- For recognising the interrupt (Ctrl + C), `console.c` was modified. `ctrlkill` function is called when input is recognised, which is defined in `proc.c`.
- This function first stores the console process (`pid` 2) in a variable. If console process is not found, then it exits with an error message. If console process exists, then it is assigned as the parent

of all the processes with `pid` > 2. This is because `initproc` once born, gives birth to console process, which then produces other children. So, assigning it as the parent ensures that when we kill processes with `pid` > 2, they don't remain as zombie. `killed` attribute of process is set to be 1 for `pid` > 2 processes.

- Correct acquiring and release of locks have been implemented in order to prevent system from entering panic mode.

```c
void ctrlkill(void)
{
  struct proc *p;
  cprintf("\nCtrl-C is detected by xv6\n");
  acquire(&ptable.lock);

  // this for-loop finds & stores the console process and throws error if not found
  struct proc *console_proc = 0;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == 2)
    {
      console_proc = p;
      break;
    }
  }

  if (!console_proc)
  {
    release(&ptable.lock);
    cprintf("Console process not found!\n");
    return;
  }

  // the for-loop kills the processes with pid > 2
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid > 2 && p->killed == 0)
    {
      p->killed = 1;
      p->forced_sleep = 0;
      p->parent = console_proc;
      release(&ptable.lock);
      wakeup(p);
      kill(p->pid);
      acquire(&ptable.lock);
    }
  }

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

0.2 SIGBG (Signal Background)

- For recognising the interrupt (Ctrl + B), `console.c` was modified. `ctrlfreeze` function is called when input is recognised, which is defined in `proc.c`.
- For this function, a new attribute of process is introduced named `forced_sleep`. This attribute is set to 1 for all processes with `pid` > 2.
- The `scheduler` function was modified to prevent process from being scheduled in case `forced_sleep` attribute is set to 1.
- According to the state of `forced_sleep`, the `all_frozen` variable is set to 0 if it is 0 and vice versa.
- Correct acquiring and release of locks have been implemented in order to prevent system from entering panic mode.

```c
void ctrlfreeze(void)
{
  struct proc *p;

  cprintf("\nCtrl-B is detected by xv6\n");
  acquire(&ptable.lock); // Lock the process table

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid > 2) // Freeze processes with pid > 2
    {
      p->forced_sleep = 1;
    }
  }
  release(&ptable.lock); // Unlock after modifying process states
}
```

```c
int wait(void)
{
    (...)
    int all_frozen = 1;

    (...)
    if (p->state == RUNNABLE && p->forced_sleep == 0)
        all_frozen = 0;
    (...)

    if (havekids && all_frozen)
    {
      // If we have children but they're all frozen,
      // allow the wait call to return to the shell
      release(&ptable.lock);
      return -2; // Special return code to indicate frozen children
    }

    (...)
}
```

0.3 SIGFG (Signal Foreground)

- For recognising the interrupt (Ctrl + F), `console.c` was modified. `ctrlrestart` function is called when input is recognised, which is defined in `proc.c`.
- In this function, for all processes, `forced_sleep` attribute is set to 0. This now allows the process to be scheduled by the scehduler and thus restarts all frozen processes.
- Correct acquiring and release of locks have been implemented in order to prevent system from entering panic mode.

```c
void ctrlrestart(void)
{
  struct proc *p;
  cprintf("\nCtrl-F is detected by xv6\n");
  acquire(&ptable.lock); // Lock before modifying state
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    p->forced_sleep = 0;
  }
  release(&ptable.lock); // Unlock after chang
}
```

0.4 SIGCUSTOM (Signal Custom)

- For recognising the interrupt (Ctrl + G), `console.c` was modified. `registerctrl()` is called once the keyboard input is recognised, which is defined in `proc.c`.

- For implementing the system call, as in previous assignment, changes were done in `syscall.h` and `syscall.c` to define the system call as `SYS_signal`. Corresponding changes were done in usys.S and user.h. `sys_signal` was implemented accordingly in `sysproc.c` and the `registerctrl` function was defined in the `proc.c`.
- In `proc.h`, new attribute for process named `sighandler` was defined which is set to the handler address by the `sys_signal` function. Now that the handler is registered, the `registerctrl` function now sets the `sig_handled` to be 1 and then the instruction pointer address is set to the `sighandler` value. This thus redirects the execution of the process to the signal handler. It jumps to `sighandler` when returning from trap.
- Correspondingly, in `trap.c`, once the signal is handled, in the default case of `trap` function, the stack pointed is now moved down by 4 bytes and allocated the old instruction pointer address to return back to previous execution after signal is handled.

```
void registerctrl(void)
{
  struct proc *p = myproc(); // Get the current process

  cprintf("\nCtrl-G is detected by xv6\n");
  if (p->sighandler) // If a handler is registered
  {
    p->sig_handled = 1;
    p->tf->eip = (uint)p->sighandler;
  }
  else
  {
    cprintf("No signal handler registered. Ignoring SIGCUSTOM.\n");
  }
}
```

```
int sys_signal(void)
{
  int handler_addr;
  if (argint(0, &handler_addr) < 0) {
      return -1;  // Invalid argument
  }
  sighandler_t handler = (sighandler_t) handler_addr;

  myproc()->sighandler = handler;
  return 0;
}
```

```
void trap(struct trapframe *tf)
{
    (...)

    if (p && p->sig_handled)
    {
      // p->tf->eip = (uint)p->sighandler;
      uint old_eip = p->tf->eip;
      p->tf->esp -= 4;
      *(uint *)(p->tf->esp) = old_eip;
      return;
    }

    (...)
}
```

xv6 Scheduler

0.5 custom_fork

- First, the `SYS_custom_fork` is implemented and `sys_custom_fork` was defined in `sysproc.c`, which then calls the `custom_fork` function in `proc.c`.

- The function is implemented similar to the `fork` function, except the new variables are added and assigned, namely `start_later_flag` and `exec_time` to support delayed start and run for a particular time only. If the `start_later_flag` is set to 0, then the new process is set as `RUNNABLE`, else it is not.

```c
int custom_fork(int start_later_flag, int exec_time)
{
  if (start_later_flag < 0 || exec_time < 0)
  {
    return -1;
  }

  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if ((np = allocproc()) == 0)
  {
    return -1;
  }

  // Copy process state from proc.
  if ((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0)
  {
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    // np->start_later = start_later_flag;
    // np->exec_time = exec_time;
    // np->start_ticks = 0;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;
  np->start_later = start_later_flag;
  np->exec_time = exec_time;
  np->start_ticks = ticks;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for (i = 0; i < NOFILE; i++)
    if (curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  if (start_later_flag == 0)
  {
    np->state = RUNNABLE;
  }

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

```c
int sys_custom_fork(void) {
  int start_later, exec_time;
  if (argint(0, &start_later) < 0 || argint(1, &exec_time) < 0)
      return -1;

  return custom_fork(start_later, exec_time);
}
```

0.6 sys_scheduler_start

- First the SYS_scheduler_start system call was implemented and `sys_scheduler_start` was defined in syspoc.c, which then calls the scheduler_start function defined in proc.c.
- This function checks that for all processes if the `start_later_flag` is set to 1 and the process is an `EMBRYO` process, then it marks the process to be `RUNNABLE` and thus ready to be scheduled.

```
int scheduler_start(void)
{
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->start_later && p->state == EMBRYO)
    {
      p->state = RUNNABLE;
      p->start_ticks = ticks; // Set the start time
    }
  }
  release(&ptable.lock);
  return 0;
}
```

```
int sys_scheduler_start(void)
{
  return scheduler_start();
}
```

0.7 Scheduler Profiler

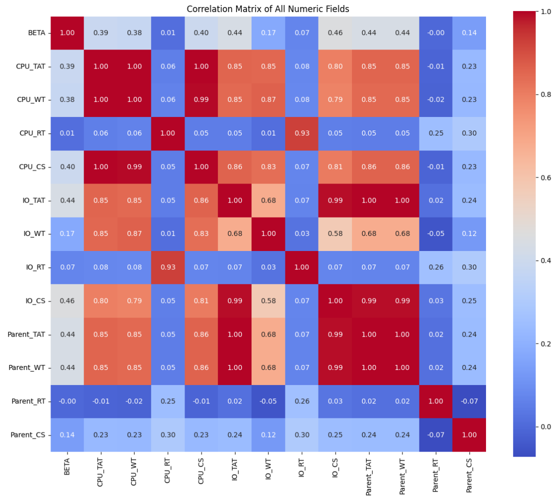
- Relevant attributes have been defined for every process in `proc.h` - `start_later`, `exec_time`, `start_ticks`, `arrival_time`, `completion_time`, `running_time`, `last_running_time`, `response_time`, `context_switch`.
- The `print_metrics` function defined in proc.c calculates the required metrics as follows:
  - `PID = p->pid`
  - `TAT = p->completion_time - p->arrival_time`
  - `WT = p->completion_time - p->arrival_time - p->running_time`
  - `RT = p->response_time`
  - `CS = p->context_switch`
- The arrival time is assigned the default value of ticks at the point of initialization. The completion time is updated as the process ends.
- The running time, response time, and the number of context switches are calculated in the `scheduler` function.
- Scoring is calculated based on the following function:
  `-ALPHA * running_time + BETA * (ticks - arrival_time - running_time)`
  This ensures that the process who have used a lot of CPU time are penalized and rewards processes who have waited longer.
- response_time is set only the first time a process runs. It records how long the process waited before starting execution. `RT = first_run_time - arrival_time`
- context_switch variable is incremented every time the scheduler switches to this process. It is the count of how many times a process was selected to run.
- When the process stops running, we update its total CPU usage. `running_time += (current_ticks - last_start_running)`.

- `scheduler` function prevents processes from running before a designated delay. Ensures that start_later processes only begin execution after a given amount of time has passed.

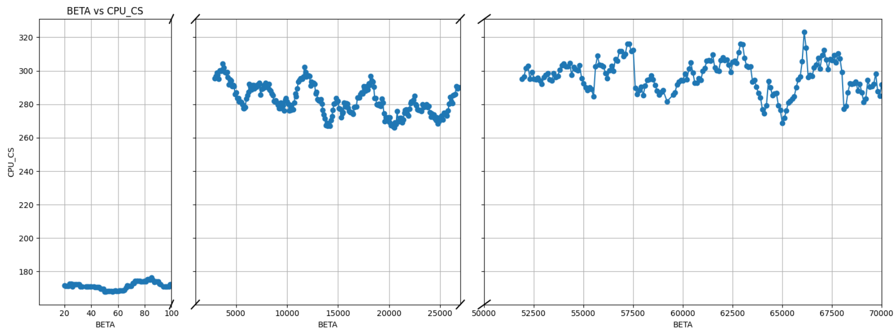
```c
void scheduler(void) {
    struct proc *p;
    struct proc *bestP = 0;
    struct cpu *c = mycpu();
    c->proc = 0;

    for (;;) {
        sti(); // Enable interrupts
        acquire(&ptable.lock);
        int best_score = -(1 << 10);

        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if (p->state != RUNNABLE) continue;
            if (p->forced_sleep) continue;
            int score_p = -ALPHA * (p->running_time) + BETA * (ticks - p->arrival_time -
                p->running_time);
            if (!bestP || score_p > best_score) {
                bestP = p;
                best_score = score_p;
            }
        }

        if (bestP && bestP->state == RUNNABLE) {
            bestP->last_start_running = ticks;
            c->proc = bestP;
            switchuvm(bestP);
            bestP->state = RUNNING;
            swtch(&(c->scheduler), bestP->context);
            switchkvm();
            bestP->running_time += ticks - bestP->last_start_running;
            c->proc = 0;
        }

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

0.8 Priority Boosting Scheduler

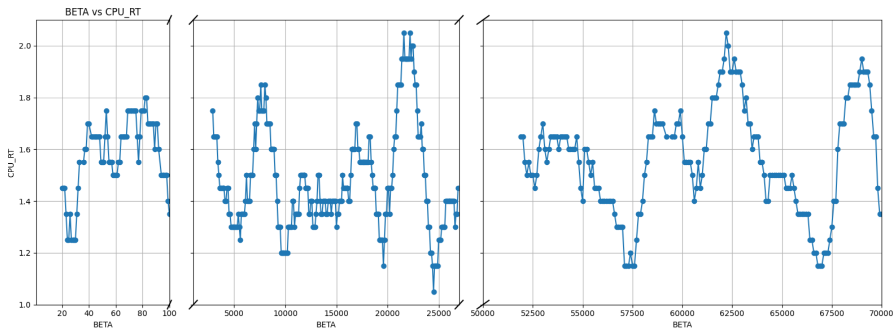
- In the given formula, we initialize the initial priority to be the same for all processes. So, in the comparison of the best score of two processes, the initial priority need not be required.
- For simplicity, assuming the initial priority to be zero, we get an expression consisting only of coefficients α and β. Here as well for comparison, if we bring out α, we are left with the ratio β/α, which is the main contributor to the value of the expression.
- So, for our experiments, we have fixed $\alpha = 1$ and iterated over different values of β.
- The following is the correlation matrix obtained between every pair of metric (including β).



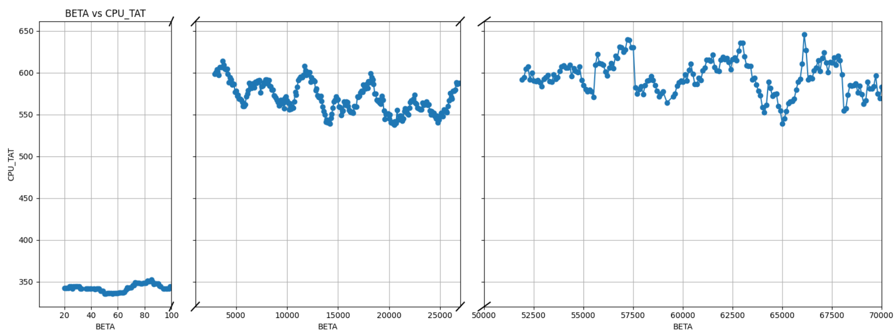
- However, not much was revealed regarding the general correlation between β.
- The value of β was then iterated over from 1-100, then from 1000-27000 and from 50000-70000 to understand the trend of the metric versus β. The following graphs were obtained for the same.
- The general trend observed is that for CPU_CS, the value increases with β upto a certain point but shows a slightly decreasing trend for very high values of β.



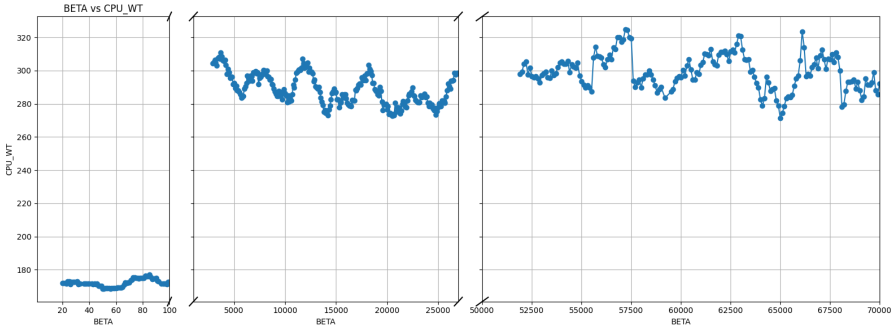
- For CPU_RT, we can observe a regular trend of ups & downs in the graph indicating cyclic behavior.



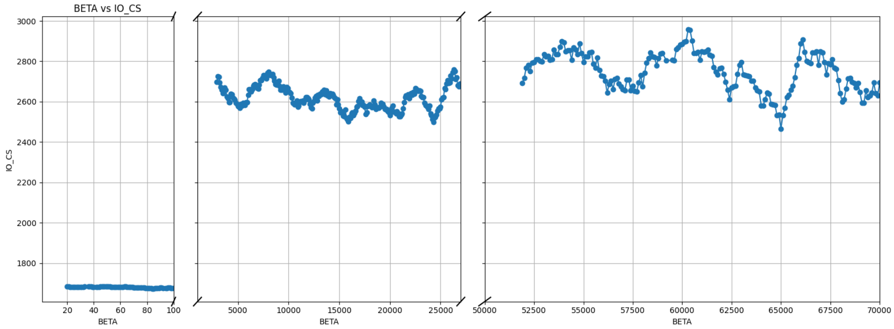
- For CPU_TAT, it follows a similar trend as CPU_CS.



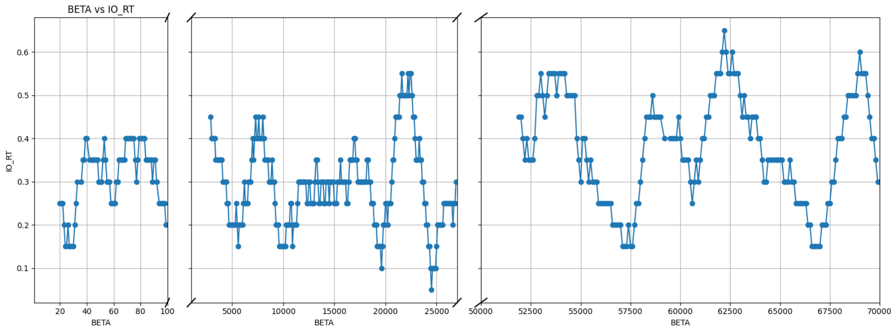
- Similarly for CPU_WT, it shows a similar trend as CPU_CS.



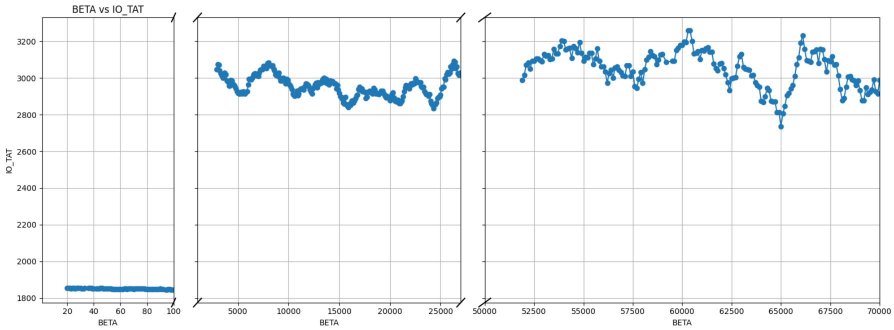
- IO_CS also shows a similar trend as CPU_CS.



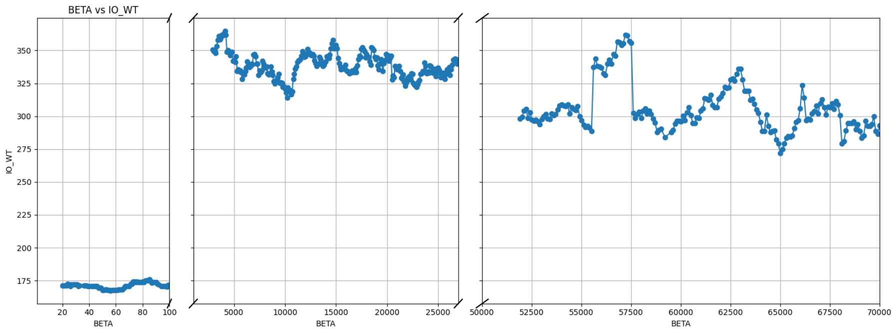
- IO_RT also follows a similar trend as CPU_RT.



- IO_TAT also shows a similar trend to IO_CS.



- IO_WT (similar trend to IO_CS) however there are more deviations at higher β and more decline in the value at higher β.



- Similar to the patterns observed for CPU and IO processes, we can draw similar conclusions for Parent process as well, though the Parent_CS shows a rather cyclic behavior with regular ups and down with a general upward trend.