COL331 Operating System
Assignment 2 Easy

Aditya Sahu (2022CS11113)
Umesh Kumar (2022CS11115)

Part 1: Signal Handling in xv6

Ctrl-C Handling

To implement signal handling for `Ctrl-C`, we modified the interrupt handler in `console.c`. When the user presses `Ctrl-C`, represented by the ASCII control character `C('C')`, the system executes a block of code designed to detect and respond to the signal.

This functionality is used to terminate all running user-level processes (except the init and shell processes with PID 1 and 2). We achieve this by printing an informative message and calling a new function `kill_user_procs()` defined in `proc.c`.

The following code was added to `console.c` to detect and respond to Ctrl-C:

```
case C('C'):{  // Detect Ctrl+C
  const char *msg = "Ctrl-C is detected by xv6\n";
  for (int i = 0; msg[i]; i++)
    consputc(msg[i]);  // Print message character-by-character
  release(&cons.lock);  // Release lock before killing processes
  kill_user_procs();    // Kill all user processes
  acquire(&cons.lock);  // Reacquire the lock after operation
  break;
}
```

Listing 1: console.c: Ctrl-C interrupt handling

The function `kill_user_procs()` is a custom kernel function implemented in `proc.c`. It iterates over the process table and kills all processes with PID greater than 2 that are in any state except `UNUSED`. This ensures the shell and init processes are not affected.

Here is the implementation of `kill_user_procs()`:

```
void
kill_user_procs(void)
{
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid > 2 && p->state != UNUSED){
      kill(p->pid);  // Send kill signal to user process
    }
  }
```

```
}
```

Listing 2: proc.c: kill_user_procs implementation

This mechanism provides a way for the OS to immediately terminate all user processes from the console, which can be useful for debugging or system recovery in a controlled environment.

Ctrl-B Handling

To implement background suspension via `Ctrl-B`, modifications were made across multiple files to coordinate detection of the input, suspension of all user-level processes, and safe context switching via the timer interrupt.

1. Detecting Ctrl-B in console.c When `Ctrl-B` is pressed, we print a message using `consputc()`, release the console lock, and set a global flag `yield_on_timer = 1`. This flag signals the timer interrupt to suspend all user processes.

```c
case C('B'):  // Ctrl+B detected
  const char *msg3 = "Ctrl-B is detected by xv6\n";
  for (int i = 0; msg3[i]; i++)
    consputc(msg3[i]);
  release(&cons.lock);
  extern int yield_on_timer;
  yield_on_timer = 1;
  return;
```

Listing 3: console.c: Ctrl-B detection

2. Handling timer interrupt in trap.c In `trap.c`, we check if the current process is running and if `yield_on_timer == 1` on each timer interrupt. If so, we call a custom function `yield2()` to suspend all user-level processes. The flag is then updated to prevent repeated suspension.

```c
if(myproc() && myproc()->state == RUNNING && yield_on_timer == 1 && tf->
    trapno == T_IRQ0+IRQ_TIMER){
  yield_on_timer = 2;
  yield2();
}
```

Listing 4: trap.c: Trigger suspension on timer interrupt

3. Suspending processes in proc.c The `yield2()` function initiates process suspension by calling `suspend_user_procs()` and then yields the CPU.

```c
void
yield2(void)
{
  suspend_user_procs();
  acquire(&ptable.lock);  // Yield lock
  if(myproc()->state != SUSPENDED)
```

```c
    myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

Listing 5: proc.c: yield2 function

The function `suspend_user_procs()` iterates through the process table and changes the state of all user processes (excluding init and shell) to `SUSPENDED`. It also ensures that the shell (PID 2) is made runnable if it was sleeping.

```c
void suspend_user_procs(void)
{
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid != 2 && p->pid != 1 && (p->state == RUNNABLE || p->state ==
     RUNNING)){
      p->state = SUSPENDED;
    }
  }
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == 2 && p->state == SLEEPING) {
      p->state = RUNNABLE;
    }
  }
  release(&ptable.lock);
}
```

Listing 6: proc.c: suspend_user_procs function

4. Modifying wait to ignore suspended processes In the `wait()` function, a condition is added to exclude suspended child processes from being considered as "alive" when checking for children.

```c
if(p->state != SUSPENDED)
  havekids = 1;
```

Listing 7: proc.c: wait() ignores suspended children

This implementation ensures that all user processes can be paused system-wide upon pressing `Ctrl-B`, allowing for clean suspension and later resumption via other commands.

Ctrl-F Handling

To complement the suspension functionality provided by `Ctrl-B`, we implemented `Ctrl-F` to resume all previously suspended user-level processes.

1. Detecting Ctrl-F in console.c When `Ctrl-F` is pressed, the system prints a confirmation message and invokes the `resume_user_procs()` function after releasing the console lock. This resumes all processes that were put into the `SUSPENDED` state.

```c
case C('F'):
  const char *msg4 = "Ctrl-F is detected by xv6\n";
  for (int i = 0; msg4[i]; i++)
    consputc(msg4[i]);
  release(&cons.lock);
  resume_user_procs();  // Resume all user processes
  return;
```

Listing 8: console.c: Ctrl-F detection

2. Resuming suspended processes in proc.c The function `resume_user_procs()` traverses the process table and restores the state of each process from `SUSPENDED` to `RUNNABLE`. This allows the scheduler to pick these processes for execution again.

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

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

Listing 9: proc.c: resume_user_procs implementation

This mechanism works seamlessly with the existing xv6 scheduler and restores the system back to a running state after global suspension triggered by `Ctrl-B`.

Ctrl-G Handling (Custom Signal Handler)

The `Ctrl-G` functionality introduces support for user-defined signal handlers in xv6. Upon detecting `Ctrl-G`, the system triggers a custom signal (`SIGCUSTOM`) for the currently running process. If the process has registered a signal handler, control is transferred to that handler in a safe and interrupt-driven manner.

1. Data Structures in proc.h We extended the `proc` structure to support custom signal handling. The added fields include:

- `custom_handler`: function pointer to user-defined signal handler.
- `pending_custom`: flag indicating a pending custom signal.
- `tf1`: backup of the current trapframe before jumping to the signal handler.

```c
typedef void (*sighandler_t)(void);

sighandler_t custom_handler;   // User-registered custom signal handler
```

```c
int pending_custom;               // Flag for pending SIGCUSTOM
struct trapframe *tf1;            // Backup trapframe for returning
```

Listing 10: proc.h: signal handler fields

2. Ctrl-G detection in console.c When `Ctrl-G` is pressed, a message is printed and, if a handler is registered, the `pending_custom` flag is set.

```c
case C('G'):  // Ctrl+G detected
  const char *msg2 = "Ctrl-G is detected by xv6\n";
  for (int i = 0; msg2[i]; i++)
    consputc(msg2[i]);
  if(myproc() && myproc()->custom_handler)
    myproc()->pending_custom = 1;
  release(&cons.lock);
  return;
```

Listing 11: console.c: Ctrl-G detection

3. Signal delivery in trap.c In the trap handler, we check for a pending custom signal and invoke the delivery function.

```c
if(myproc() && myproc()->pending_custom && myproc()->custom_handler != 0
   && !myproc()->killed) {
  handle_pending_custom_signal(tf);
}
```

Listing 12: trap.c: invoking handler

4. Delivering the custom signal (proc.c) The function `handle_pending_custom_signal()` backs up the current trapframe, stores the return instruction pointer, and redirects execution to the handler.

```c
void handle_pending_custom_signal(struct trapframe *tf)
{
  struct proc *p = myproc();
  if(p == 0)
    return;

  if(p->pending_custom && p->custom_handler != 0 ){
    p->pending_custom = 0;
    if(!p->tf1){
      p->tf1 = (struct trapframe*)kalloc();
    }
    memmove(p->tf1, tf, sizeof(struct trapframe));  // Save state
    p->prev_eip = tf->eip;                           // Save return point
    tf->eip = (uint)p->custom_handler;              // Jump to handler
  }
}
```

Listing 13: proc.c: handle custom signal

5. Registering the handler (sysproc.c) We implemented a new system call `signal()` to register the user-level handler.

```c
int sys_signal(void)
{
  sighandler_t handler;
  if(argptr(0, (void*)&handler, sizeof(handler)) < 0)
    return -1;
  myproc()->custom_handler = handler;
  return 0;
}
```

Listing 14: sysproc.c: sys_signal syscall

6. Returning from the handler To ensure a clean return from the user-level handler, the original trapframe is restored on certain traps (like page faults or general protection faults), allowing the process to continue from where it was interrupted.

```c
case T_PGFLT: {
  if(myproc() == 0) break;
  if(myproc()->killed) exit();
  memmove(myproc()->tf, myproc()->tf1, sizeof(struct trapframe));
  break;
}
case 13: {
  if(myproc()->killed) exit();
  memmove(myproc()->tf, myproc()->tf1, sizeof(struct trapframe));
  break;
}
```

Listing 15: trap.c: restoring trapframe

This mechanism allows xv6 to mimic user-level signal handling by safely backing up the processor state and redirecting execution flow to user-defined functions, a crucial step toward supporting asynchronous user-space event handling.

Part 2: xv6 Scheduler

We extended the scheduler to implement dynamic priority scheduling. Each process maintains:

- `initial_priority`, `dyn_priority`
- `cpu_ticks`, `wait_ticks`, `context_switches`
- `creation_time`, `first_run_time`, `finish_time`

Dynamic priority is updated as:

```c
p->dyn_priority = p->initial_priority - (ALPHA * p->cpu_ticks) + (BETA * p
    ->wait_ticks);
```

Scheduling logic in `proc.c` chooses the process with highest dynamic priority. Ties are broken using PID.

```
if (highest == 0 || p->dyn_priority > highest->dyn_priority ||
    (p->dyn_priority == highest->dyn_priority && p->pid < highest->pid)) {
  highest = p;
}
```

Effect of α and β

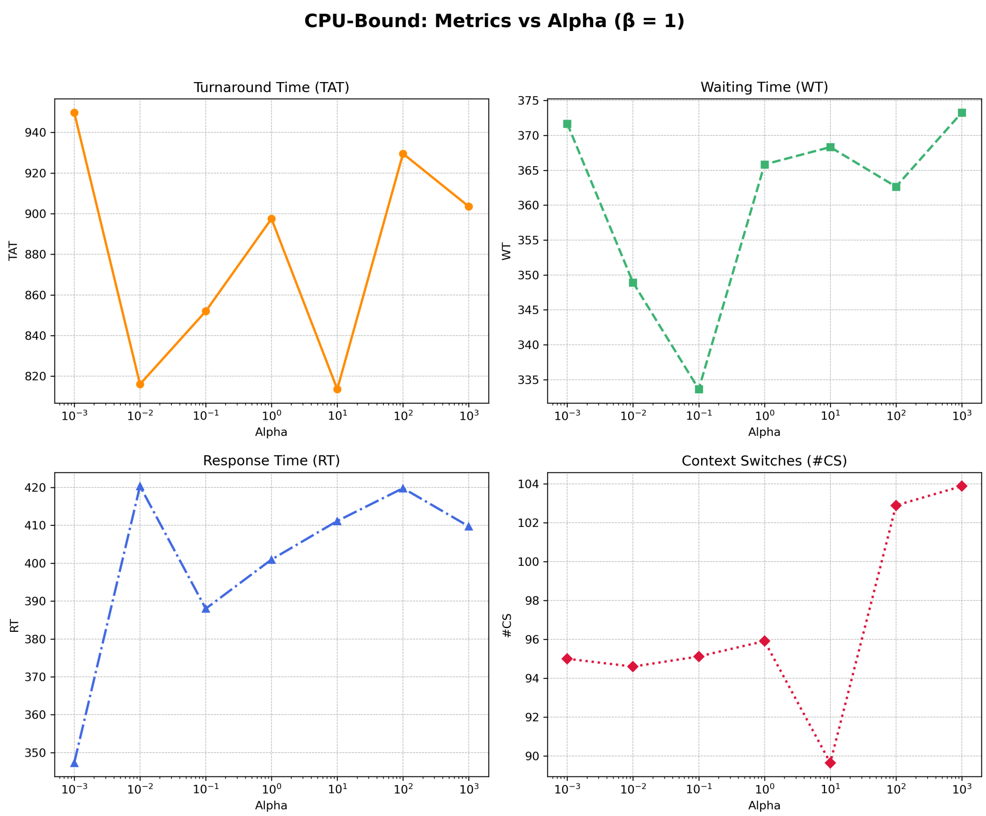

Figure 1: CPU-Bound: Metrics vs Alpha ($\beta = 1$)

Discussion: As α increases, CPU-bound processes receive lower effective priorities the more CPU time they accumulate. Consequently:

- **Turnaround Time (TAT)** tends to increase because these CPU-bound jobs lose scheduling priority faster.
- **Waiting Time (WT)** also rises as they are preempted more frequently in favor of processes with lower CPU usage.
- **Response Time (RT)** exhibits a modest increase, reflecting longer initial delays before service.

- **Context Switches (CS)** go up since the scheduler often preempts CPU-bound tasks when α is higher.

Overall, a larger α means the scheduler deprioritizes CPU-intensive tasks more aggressively, hurting their performance metrics.

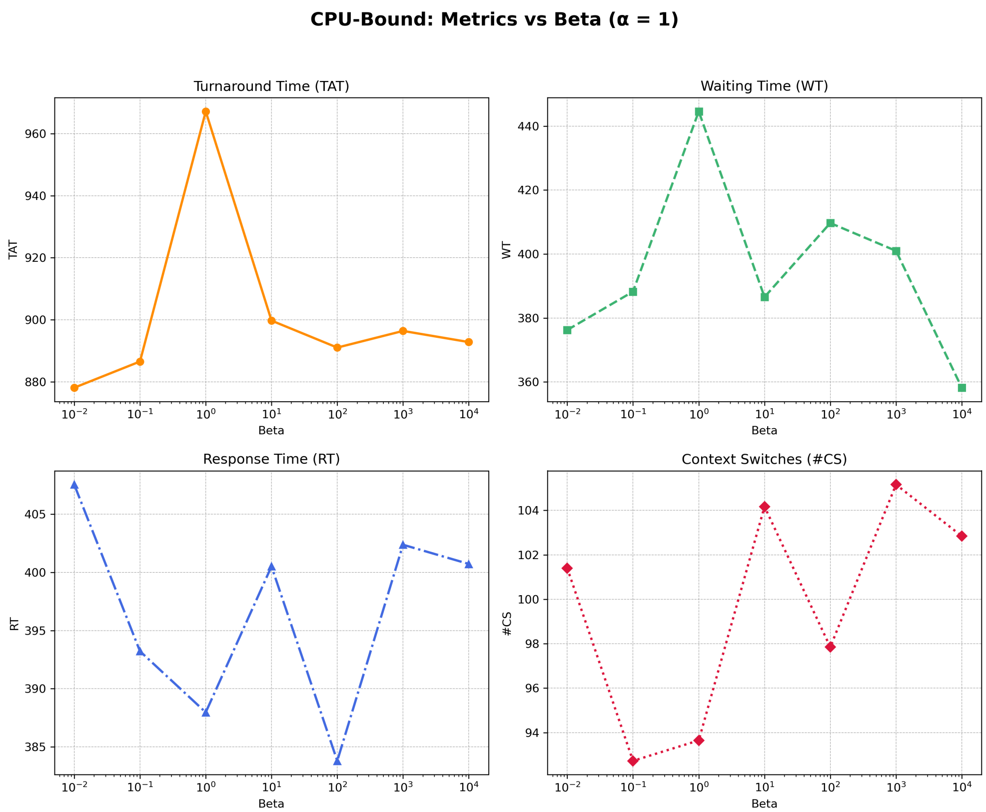

Figure 2: CPU-Bound: Metrics vs Beta (α fixed)

Discussion: For CPU-bound processes, as β increases:

- **Turnaround Time (TAT)** shows a slight decrease, indicating a modest improvement in overall job completion time.
- **Waiting Time (WT)** also decreases slightly, which suggests that processes spend less time in the ready queue.
- **Response Time (RT)** exhibits little change, showing that the initial responsiveness is not significantly affected.
- **Context Switches (CS)** increase, implying that higher β values result in more frequent scheduling events.

This trend demonstrates that raising β helps in marginally favoring CPU-bound tasks by reducing their waiting and turnaround times but at the cost of increased scheduling overhead.

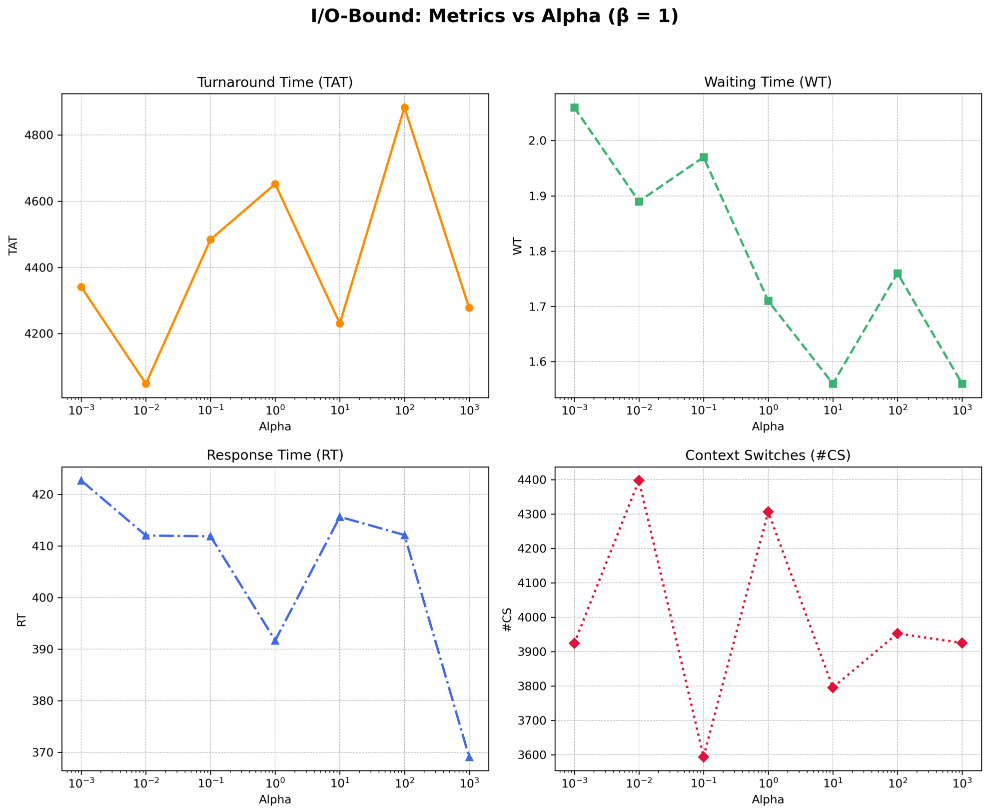

Figure 3: I/O-Bound: Metrics vs Alpha ($\beta = 1$)

Discussion: For I/O-bound processes, an increase in α typically results in:

- **Turnaround Time (TAT):** Decreases as I/O-bound processes are less penalized for CPU usage.
- **Waiting Time (WT):** Also decreases since these processes spend less time waiting in the ready queue.
- **Response Time (RT):** Improves (i.e., decreases), reflecting quicker initial scheduling.
- **Context Switches (CS):** Increase due to more frequent preemptions imposed by the scheduler.

Overall, a higher α favors I/O-bound tasks, allowing them to complete faster while incurring more scheduling overhead.

Discussion: For I/O-bound processes, as β increases:

- **Turnaround Time (TAT):** Decreases, indicating improved completion times as waiting time is reduced.
- **Waiting Time (WT):** Decreases, meaning processes spend less time in the ready queue.

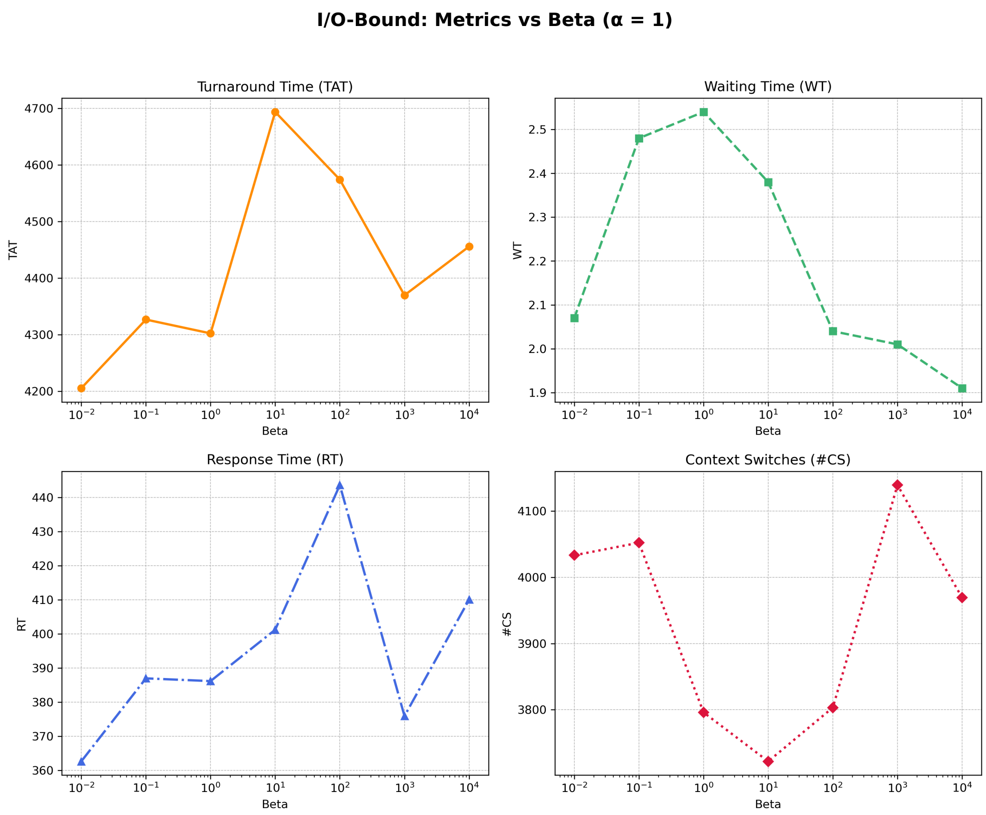

Figure 4: I/O-Bound: Metrics vs Beta (α fixed)

- **Response Time (RT):** Also decreases, leading to quicker initial response.
- **Context Switches (CS):** Increase, showing that the scheduler is more active in preempting and rescheduling processes.

This suggests that a higher β makes the scheduler more responsive for I/O-bound tasks by reducing delays, although it results in a higher rate of context switching.