REPORT:ASSIGNMENT 2(EASY)

OPERATING SYSTEM

JITESH VIVEK SHAMDASANI 2024JCS2043	**VIKASH PANDEY 2024JCS2613**

Signal Handling in xv6

1. Keyboard Event Detection and Initial Handling

- **Console Interrupt Handler (consoleintr):**
  When a keyboard event occurs, the function reading input in the console interrupt handler (located in *console.c*) continuously calls the character getter (via getc()). For each character, a switch-case block distinguishes the control codes.

    - **Ctrl+C (SIGINT):**

        - The control code is recognized as C('C').

        - A message ("Ctrl-C is detected by xv6") is printed immediately.

        - The code then acquires the process table lock and iterates over all processes. For each process with a pid greater than 2 (excluding the init and shell processes), the killed flag is set.

        - When the flagged process eventually reaches a check in the trap handler (or scheduler), it will be terminated.

    - **Ctrl+B (SIGBG):**

        - The keypress for Ctrl+B is detected (through C('B')) and a dedicated flag bit (FLAG_CTRLB) is set in a local 8-bit variable.

        - After processing input, a conditional block checks if this bit is set, then prints an announcement ("Ctrl-B is detected by xv6") and, while holding the process table lock, iterates over processes with pid greater than 2. For these processes, the background flag (b) is set, thereby marking them as suspended.

        - Within the scheduler, there is a condition (e.g., if(p->b == 1)) to skip scheduling these processes.

    - **Ctrl+F (SIGFG):**

        - Similarly, the console handler identifies Ctrl+F (C('F')) and sets a flag bit (FLAG_CTRLF).

        - Although the provided code snippet shows only the flag being set, further processing (either in a separate block or later in the code) would check this flag and clear the suspension flag (b) for processes, thus making them runnable again.

    - **Ctrl+G (SIGCUSTOM):**

- Upon detecting Ctrl+G (C('G')), a corresponding message is printed.
- The handler then calls a helper function (send_sigcustom()), which marks the current process by setting its pending_signal field.
- This marks the process as needing to handle the custom signal.

```
  case C('B'):
  flags |= FLAG_CTRLB;  // set bit for Ctrl+B
    break;


case C('F'):
flags |= FLAG_CTRLF;  // set bit for Ctrl+F
  break;

////////////////////
/* C +   G */
case C('G'):
char* ctrl_g_msg="\nCtrl-G is detected by xv6\n";

  while (*ctrl_g_msg!='\0')
  {
    consputc(*ctrl_g_msg);
    ctrl_g_msg++;
  }
  send_sigcustom();  // Defined in proc.c (see next step)
  break;
////////////////////////////
```

2. Signal Delivery via the Trap Handler

- **Trap Handler (trap.c):**
  After the console interrupt has completed its tasks and just before returning control to user space, the trap handler checks if the current process has a pending signal and a non-null signal handler registered.
    - **For SIGCUSTOM (Ctrl+G):**
        - If both pending_signal and the registered signal_handler are set on the process, the handler proceeds to deliver the signal.
        - The process's current trapframe (the CPU register state) is saved by allocating kernel memory and performing a memory copy (using memmove) into a new field (saved_tf) in the process structure.
        - A special magic constant (MAGIC_SIGRET, set to 0xDEADBEEF) is then pushed onto the user stack. This action is done by decrementing the stack pointer (esp) and copying the magic value to that location.
        - The trap frame's instruction pointer (eip) is set to the address of the registered signal handler.
        - The pending flag (pending_signal) is then cleared so that the signal is delivered only once.
- **Fault Handling for Signal Return:**
  After the custom signal handler in user space finishes its execution, when it executes the RET instruction, the magic constant is popped off the stack as the return address. Since

0xDEADBEEF does not correspond to a valid executable address, this action causes a page fault (or general protection fault).

- In the fault handling routine within *trap.c*, the faulting address is examined.
- When the faulting address matches the magic value, the kernel interprets this as an indication that the signal handler is returning.
- The kernel then restores the saved trapframe (using the data stored in saved_tf) back into the process's active trapframe.
- The allocated memory for the saved trapframe is freed, effectively completing the return from the signal handler without requiring an explicit user-level sigreturn stub.

3. System Calls and Process Structure Modifications

- **Signal Registration (sys_signal):**
  - The system call sys_signal allows a process to register a custom signal handler.
  - It accepts a single argument: a pointer to the user-defined handler function.
  - The process's structure (defined in *proc.h*) includes a field for the signal handler (signal_handler) that is updated by this system call.
- **Signal Return (sys_sigreturn):**
  - The system call sys_sigreturn is responsible for restoring the saved trapframe when a signal handler returns.
  - This syscall is invoked implicitly when the fault handler in *trap.c* recognizes the magic constant (after a RET from a signal handler).
  - It copies the saved trapframe back to the active one, frees the allocated memory, and clears the saved pointer.
- **Process Structure Enhancements:**
  - The process structure is augmented with three new fields:
    - pending_signal (an integer flag to indicate if a SIGCUSTOM is pending).
    - signal_handler (a pointer to the registered handler function).

        - saved_tf (a pointer used to store a copy of the trapframe when the signal is delivered).

4. Scheduler Interaction and Process State Management

- **Background Suspension (SIGBG / Ctrl+B):**
    - The console handler sets the background flag for processes (via the field b) when Ctrl+B is pressed.
    - In the scheduler, a check on this flag (e.g., if(p->b == 1)) determines whether a process is runnable.
    - A suspended process is not scheduled and remains inactive until a foreground signal (SIGFG, Ctrl+F) clears its background flag.
- **Foreground Reactivation (SIGFG / Ctrl+F):**
    - When Ctrl+F is pressed, the corresponding flag is set.
    - Processing in the system (or another dedicated function) then clears the background flag for processes that were previously suspended.
    - Once cleared, the process becomes eligible for scheduling again.
- **Process Termination (SIGINT / Ctrl+C):**
    - For Ctrl+C, the console interrupt handler sets the killed flag on each eligible process.
    - During subsequent trap or scheduling checks, when a process's state is examined and the killed flag is found, the process exits, thereby terminating its execution.

5. Abstracted Implementation Details and Code Flow Summary

- **Local Flag Variable:**
    - A single 8-bit variable is used in the console interrupt routine to store indicators for Ctrl+B and Ctrl+F key events.
    - The bits corresponding to these flags allow for simple later checking and handling.
- **Global Process Table and Synchronization:**
    - Critical sections modifying process states (e.g., setting the killed flag or b flag) are protected by acquiring the process table lock.

    - This prevents race conditions during modifications of the process table.

- **Magic Return Technique:**

    - Instead of depending on a linker-script–dependent sigreturn stub, the design uses a magic constant (0xDEADBEEF) to signal when the signal handler returns.

    - The fault handler (for page faults or protection faults) intercepts attempts to jump to this magic value and triggers the restoration of the original process context.

- **Seamless Signal Invocation:**

    - For SIGCUSTOM, if a user has registered a handler, the kernel changes the process's execution context to directly begin executing that handler.

    - If no handler is registered, the custom signal is effectively ignored.

- **Process Context Saving and Restoration:**

    - Before diverting execution to a signal handler, the entire trapframe is saved so that execution can be accurately resumed afterward.

    - Upon detecting the return from the signal handler (via the magic constant), the saved state is restored so that the process continues as if interrupted only temporarily.

6. Observation

- **Key Press Feedback:**

    - Each key event (Ctrl+C, Ctrl+G, etc.) is immediately echoed to the console. This immediate feedback is helpful for debugging and ensuring that the kernel is processing inputs promptly.

- **Separation of Signal Types:**

    - The design cleanly separates actions among termination (SIGINT), suspension (SIGBG), foreground reactivation (SIGFG), and custom signal handling (SIGCUSTOM).

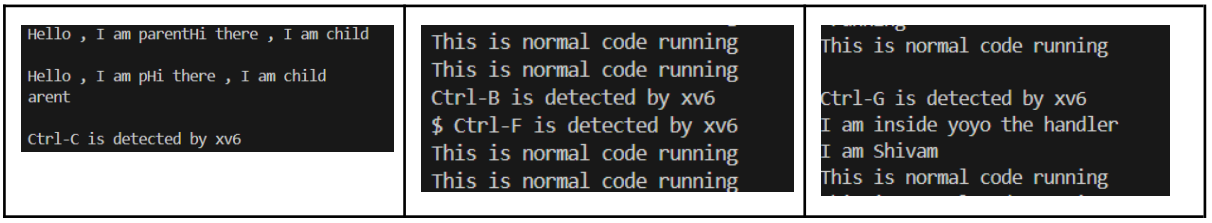
    - This separation allows for independent modifications and debugging of individual signal behaviors.

- **Fault Handling Integration:**

- By leveraging the existing fault handler in *trap.c*, the implementation avoids having to add a dedicated return path for signals, thereby simplifying the design.

- **Minimal Interface Changes in User Space:**
  - The user test program interface remains unchanged. The new system calls (signal and sigreturn) have wrappers in the user assembly files and declarations in *user.h*, ensuring that the test program can register a handler without knowing about the underlying magic constant or fault handling.

- **Robustness Considerations:**
  - The code includes checks for memory allocation failures (e.g., when copying the trapframe) and outputs error messages to assist in diagnosing faults.



---

xv6 Scheduler

IMPLEMENTATION DETAILS:

1. Custom Fork Implementation

Control Flow & Implementation Details

Process Creation and State Copying:

- The custom fork is implemented via custom_fork(int start_later, int exec_time).
- A new process is created using allocproc() and initialized as follows:
  - Virtual memory is copied using copyuvm().
  - Trapframe and file descriptors are duplicated from the parent.
  - %eax is set to 0 in the child to mimic standard fork() behavior.

Marking Custom Process Fields:

- custom = 1 marks the process as custom forked.

- start_later controls delayed scheduling. If 1, state is set to CUSTOM_WAIT.
- exec_time is the maximum number of ticks the process should run. If -1, it is unlimited.
- Scheduler-related fields are initialized:
    - creation_time, first_cpu_time = -1, rtime = 0, context_switches = 0.
- If start_later == 0 and exec_time == -1, it behaves like normal fork().

System Call Wrapper:

- sys_custom_fork() reads user arguments, invokes custom_fork(), and returns the PID.

2. sys_scheduler_start System Call

Control Flow & Implementation Details

Purpose:

- Allows deferred scheduling of processes created using custom fork with start_later = 1.

Operation:

- Iterates through the process table.
- All processes in CUSTOM_WAIT state are transitioned to RUNNABLE.
- If no such processes exist, the call is a no-op.

3. Yield Mechanism and Execution Time Enforcement

Control Flow & Implementation Details

Scheduler Function Modifications for kill:

- Modified yield() checks for custom processes with bounded execution time.
- If a custom process has exec_time != -1 and rtime >= exec_time, it is terminated using exit().
-

```
if (chosen->exec_time > 0) {
  if (chosen->rtime >= chosen->exec_time) {
    chosen->killed = 1;
  }
} else {
  // skip: no execution limit or runs indefinitely
}
```

Regular Yield Behavior:

- If not exiting, the process is marked RUNNABLE, and a context switch (swtch()) is triggered.
- Timestamps are used to compute CPU burst duration (rtime is updated).
- context_switches is incremented on each yield.

4. Scheduler with Priority Boosting

Control Flow & Implementation Details

Dynamic Priority Computation:

- The scheduler chooses among RUNNABLE processes based on the formula:

$$\pi_i(t) = \pi_i(0) - \alpha \cdot C_i(t) + \beta \cdot W_i(t),$$

  - πi(0) is from INIT_PRIORITY in the Makefile.
  - rtime is accumulated CPU time.
  - waiting_time = ticks - creation_time - rtime.
  - α and β are passed via Makefile macros ALPHA and BETA.

Selection Process:

- Backgrounded processes (b == 1) are skipped.
- Highest dynamic priority is selected; ties are broken using lower PID.

Context Switch and Runtime Accounting:

- On first run, first_cpu_time is recorded.
- During execution, CPU time is tracked and added to rtime.
- After yielding, context_switches is incremented.

<u>Effects of α (ALPHA) and β (BETA) Parameters</u>

Impact on Scheduler Behavior and Profiling Metrics

- **Role in Dynamic Priority Calculation:**

$$dyn\_priority = init\_priority - \alpha \cdot rtime + \beta \cdot waiting\_time$$

  - **alpha Parameter:**
    - This parameter penalizes processes based on the amount of CPU time they have consumed. A higher value foralpha(α) causes CPU-bound tasks (which accumulate higher rtime) to see a larger reduction in their dynamic priority.
    - In terms of profiling metrics, a highalpha(α) could lead to longer turnaround times (TAT) and waiting times (WT) for CPU-bound processes because they

will be selected less often after significant CPU usage.

- Conversely, ifalpha(α) is small, the penalty for CPU consumption is minimized. This can benefit CPU-bound jobs but might lead to starvation of processes that wait longer.

- **beta Parameter:**

  - beta(β) provides a positive boost based on a process's waiting time. Higher beta(β) values result in a more significant increase in dynamic priority for processes that have waited for longer durations.

  - A larger beta(β) tends to lower waiting times (WT) and improve response times (RT) for interactive or I/O-bound tasks, as processes that have waited longer are given higher priority and thus scheduled sooner.

  - However, if beta(β) is set too high, processes that have waited very long could preempt even those that have substantial CPU usage, potentially increasing the number of context switches (#CS) if the scheduler continually reorders priorities.

Balancing Effects on Profiling Metrics:

- **Turnaround Time (TAT):**
  A highalpha(α) value may increase TAT for processes that use the CPU extensively because they are penalized. A well-tuned beta(β) can counterbalance this effect by ensuring that waiting processes eventually achieve high priority.

- **Waiting Time (WT):**
  With a higher beta(β), processes that are idle or waiting will experience a boost in their dynamic priority, reducing their overall waiting time. This parameter is critical in preventing starvation.

- **Response Time (RT):**
  Higher beta(β) values typically result in a decreased response time for processes, especially for tasks that are interactive or I/O-bound, since they obtain a priority boost sooner.

- **Context Switches (#CS):**
  Both parameters affect how frequently processes are preempted. An overly aggressive penalty (high alpha(α)) or an overly generous boost (high beta(β)) might destabilize the scheduler, leading to increased context switches if processes are frequently reprioritized. Ideally, the parameters are chosen to provide a balanced scheduling order that minimizes unnecessary context switches while keeping both CPU-bound and I/O-bound tasks responsive.