xv6 - With Strace and various Scheduling Algorithm

xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix Version 6 (v6). xv6 loosely follows the structure and style of v6, but is implemented for a modern RISC-V multiprocessor using ANSI C.

Requirement 1: System Call

System call/TRAP/Interrupt Execution cycle

- whenever any system call/TRAP occured then below cycle will be followed.
- **Cycle** : usys.S -> trampoline.S -> trap.c(user trap()) -> syscall.c -> sysproc.c -> proc.c -> trap.c (usertrap return) -> trampoline.S
- **usys.S** -> It contains assembly code which load system call number into a7 register and call environment call process(system call process).Also stores arguments inside registers like a0,a1,a2,a3...
- **trampoline.S** -> Its contain assembly code which initialize and load PCB for any process and store the content into trapeframe and iniialize all the registers.Inside this file user trap function's address is hardcoded so it will be called after this.
- **trap.c** -> All the trap/interupt handling is done in this file.This file will identify the type of trap and act accordingly, If the trap type is system call then it will call syscall() function.
- **syscall.c** -> All the system calls are passed through this systemcall() function. It will get the value of system call number from a7 register and call related system call routine.
- **sysproc.c/sysfile.c** -> This file contains all the prototypes of all system calls. System call will be redirected from this to its core execution part ehich is stored inside procc.c file.
- **proc.c** -> This file contain all the to do work of system calls.
- After all this process result will be again stored into a0 register.
- And call will be returmed to trap.c then tramploline.S and Then to usys.S, inside which ret statement has written.

Implementation of "trace" system call with "strace" user program

- trace system call will trace all the system calls which are being called during the execution of given user command.
- It will trace only those system call whoes system call number will match with user input mask.

Execution

- `- make clean`
- `- make qemu`
- `- strace [mask] [user program command]`
- `1. strace 32 grep hello README`
- `2. strace 2147483647 grep hello README`

Approach and Implementation

Changes being made in files :

- Makefile - give user program file to compile

1. **user side :**

- strace.c - contain user program of strace.
- user.h - function declaration at user side of trace system call.
- usys.pl - entry of trace system call which will be converted to assembly code at run time.

2. **kernal side :**

- defs.h - function declaration at karnal side of trace system call.
- syscall.h - define system call number to trace system call with 22.
- proc.h - define trace mask field in structure block of process.
- syscall.c - Printing details of desired system call.
- sysproc.c - prototype of trace system call.
- proc.c - routine of trace system call.

Implementation

- **strace.c** file contain user program of strace. In which trace system call is being called and when trace executed successfully, we are calling user input command useng exec.
- Retriving system call number from a7 register.
- trace system call(**proc.c**) will store user input mask inside structure of process(**proc.h**).

- We are copying this user input mask in all child being called by exec(user command).Every child is created inside fork system call. Inside that we are copying parent trace mask into child's trace mask. and aslso initializing tracemask to 0 whwnever new process is created.
- Now whenever any system call is being called it must pass through syscall function of **syscall.c**.
- Retriving user input mask from a0 register.
- By doing left shift of trace mask stored inside process's structure by system call number times, we are deciding whether this system call's info is desiered by user or not ?
- If yes then we are printing all info of that system call.
- We are maintianing all system call names and system call arguments list inside 2 arrays.
- We are printing inforamtion of all arguments being used by system call from trapeframe of that process.Accessing it's value from registers a0...a7 stored inside trapeframe.

Requirement 2: Scheduling

- Round Robin = DEFAULT (Default scheduler implemented by xv6)
- First Come First Serve = FCFS (First Come First Serve Scheduler)
- Priority Based = PBS (Priority Based Scheduler)
- Multilevel Feedback Queue = MLFQ (Multilevel Feedback Queue scheduling)

Common functionalities for all scheduler

Changes being made in files :

1. **kernal side :**

- Makefile - SCHEDULER - a macro for the compilation of the specified scheduling algorithm.
- proc.h - define time calculation fields in structure block of process.
    - cpuRunTime : Time when proces run in cpu
    - creationTime : Time at creation of process
    - endTime : Time when process ends
    - noOfTimesGotCpu : No of times when process comes in cpu
- trap.c - calling updateTime() at each clockintr().
- proc.c - updateTime() - function, time allocation at time of allocation of process and store end time at time of exit.

    - updateTime() -> This will update cpuRuntTime of all RUNNING state processes.
    - At time of new process allocation we will set below fields of process structure.
        - cpuRunTime = 0
        - endTime = 0
        - creationTime = ticks
        - noOfTimesGotCpu = 0
    - And at time when process complete its execution then we will set end time of process to current time inside exit(). -Whenever any process completes its execution or timer interrupt occur in case of premption or any kind of interrupt occur(yield),then kernal will call sched function.
- Inside sched function we are passing process context to cpu context using setch().
- Now Scheduler will be called from this.And scheduler will select next process according to the algorithm.

FCFS Scheduler

- Implement a policy that selects the process with the lowest creation time (creation time refers to the tick number when the process wascreated). The process will run until it no longer needs CPU time.

Execution

- `- make clean`
- `- make qemu SCHEDULER=FCFS`

Approach and Implementation

Changes being made in files :

1. **kernal side :**

- proc.c - FCFS scheduler routine inside schedduler().

Implementation

- Scheduler will pick a process which have lowest creation time and must be in runnable state.
- Preemption of the process after the clock interrupts is disabled in trap.c kernaltrap() and usertrap().Hence once procees got the cpu then it will release it once it done with the task.

PBS Scheduler

- Implement a non-preemptive priority-based scheduler that selects the process with the highest priority for execution.

Execution

- `- make clean`
- `- make qemu SCHEDULER=PBS`

Approach and Implementation

Change being made in files :

1. **user side :**

- setpriority.c - contain user program of setpriority.
- user.h - function declaration at user side of set_priority system call.
- usys.pl - entry of set_priority system call which will be converted to assembly code at run time.

2. **kernal side :**

- defs.h - function declaration at karnal side of set_priority system call.
- syscall.h - define system call number to trace system call with 24.
- proc.h - define time calculation fields in structure block of process.
    - sleepStartTime - When the process was last put to sleep
    - sleepTime - The sleeping time since it was last scheduled
    - staticPriority - The static priority of the process
    - DEFAULT_STATIC_PRIORITY 60
- sysproc.c - prototype of set_priority system call.
- proc.c - routine of set_priority system call and scheduler algorithm.

Implementation

- Default priority for every process is 60. User can change it using set_priority system call.
- If the new priority is lower(in terms of number) than yield interupt call will be made. Which will force scheduler to choose process again.
- Scheduler will pick the process according to the priority.
- Using niceness we are calculating each process's priority.Process which remain in sleep for longer time will get the high priority.
- niceness = (sleepTime / sleepTime+cpuRunTime)*10

- new Priority = max(0,min(staticPriority-niceness+5,100))
- After choosing the process scheduler will call swtch().

MLFQ Scheduler

- Implement a simplified preemptive MLFQ scheduler that allows processes to move between different priority queues based on their behavior and CPU bursts.
- The time-slice are as follows:
    1. For Priority 0:1 timer tick
    2. For Priority 1:2 timer ticks
    3. For Priority 2:4 timer ticks
    4. For Priority 3:8 timer ticks
    5. For Priority 4:16 timer ticks

Execution

- `- make clean`
- `- make qemu SCHEDULER=MLFQ`

Approach and Implementation

Change being made in files :

1. **kernal side :**

- proc.h - define time calculation fields in structure block of process.
    - entryTimeInCurrentQ - Entry time in the current queue
    - qTicks[5] - Number of ticks done in each queue
    - currentQ - Current queue number of the process
- proc.c - scheduler algorithm.
- trap.c - inside kernaltrap() or usertrap() if interrupt occur then prempting the current process.

Implementation

- There is no any physical implemetation of queues.Storing all the queue related information in process structure. SO its just a logical convention.
- When process is created(allocated) at that time seting currentQ as 0, entryTimeInCurrentQ and qTicks of all queue to 0.

- If any process's waiting time in current queue exceeds 16 ticks then it will be promoted to upper queue, so its priority will be increased.
- After this aging of each RUNNABLE process, scheduler will select one process.
- Scheculer will start looking for RUNNABLE process from q0 to q4 queues.If tie happens between two processes then whoes entry time is less will be choosen.
- Then scheduler will schedule the process using swtch().
- If timer interrupt occur and if process's given time frame is over then prempting the process using yield(). and then scheduler will again get the chance to schedule the process.
- devintr() function will tell us which kind of interrupt occur, using this we are identifying timer interrupt
    - returns 2 if timer interrupt
    - returns 1 if other device
    - returns 0 if not recognized

Requirement 3: procdump

- procdump is a function that is useful for debugging. It prints a list of processes to the console when a user types ctrl-p on the console

Execution

- `- make clean`
- `- make qemu`
- `- ctrl+p`

Approach and Implementation

Changes being made in files :

1. **kernal side :**

- proc.c - Modification in procdump().
- proc.h - All data is stored in process stucture.

Implementation

- process ID -> pid

- priority -> For PBS and MLFQ
- state
- running time -> cpuRunTime
- wait time -> endTime - creationTime - cpuRunTime
- nrun -> noOfTimesGotCpu
- Time spend in queues (for MLFQ)

Requirement 4: Report

- Include the performance comparison between the default and 3 implemented scheduling policies in the document by showing the average waiting and running times for processes.

Execution

- `- make clean`
- `- make qemu`
- `- schedulertest`

Approach and Implementation

Changes being made in files :

- Makefile - give user program file to compile

1. **user side :**

- schedulertest.c - Calls waitx system call to measure diffrent processes waiting and running time.
- user.h - function declaration at user side of waitx system call.
- usys.pl - entry of waitx system call which will be converted to assembly code at run time.
- time.c - to call waitx system call.

2. **kernal side :**

- defs.h - function declaration at karnal side of waitx system call.
- syscall.h - define system call number to waitx system call with 23.
- sysproc.c - prototype of waitx system call.
- proc.c - routine of waitx system call.

Implementation

- **schedulertest.c** in this file we are creating 10 processes using fork to analyze diffrent differnet scheduler average cpu running time and waiting time using waitx system call.
- waitx system call will return value of process's cpur run time and waiting time.(wait time = end time - creation time - cpu run time)
- waitx system cal will work same as wait system call. It will wait for its all children and return desired times.

Performance comparision between schedulers

- Created 10 processes. Other than fcfs, for every scheduler we are sending 5 processes for sleep. Below is the average run time and average waiting time of different schedulers: (Also set priority to 80 for PBS)
- File : **schedulertest.c**
- **Round-robin (default)**
    - Average running time: 80 ticks
    - Average waiting time: 180 ticks
- **First-come first-serve**
    - Average running time: 201 ticks
    - Average waiting time: 244 ticks
- **Priority-based**
    - Average running time: 96 ticks
    - Average waiting time: 138 ticks
- **Multi-level feedback queue**
    - Average running time: 94
    - Average waiting time: 164

Observation :

- **FCFS :**
    - As FCFS is non premptive its average waiting time will be highest.
    - Because while running one process all other will be in waiting state.
    - In worst case suppose higher cpu bound process comes first and got the cpu then all other processe have to wait. Even thogh some process need cpu for very small amount of time.
- **Round Robin :**
    - In general cases round robin performs best among these scheduler.
    - Becasue every process gets an equal share the of the cpu.

    - So It will not treat any process based on their cpu bound.
- **PBS :**
    - This is also non premptive but it favors higher priority processes.So its behaviour of average waiting time and running time is not predictable.
    - Because in some case higher cpu bound process's priority can be high and if it will get the cpu then average waiting time will increase.
- **MLFQ:**
    - It will provide higher priority to upper queues processes.
    - And also aging concept will help the processes which have longer waiting time will be moved to higher priority queue.
    - So eventually every process got the chance to get the cpu. So beside some special case of MLFQ will perform well than FCFS.