CS 354 Spring 2024

Lab 2: Monitoring Process Behavior and Dynamic Priority Scheduling [260 pts]

Due: 2/9/2024 (Fri.), 11:59 PM

1. Objectives

The objective of this lab is to extend XINU's fixed priority scheduling to dynamic priority scheduling that facilitates fair sharing of CPU cycles among processes. Through monitoring, we aim to compensate I/O-bound processes that voluntarily relinquish CPU before depleting their time slice by improving responsiveness compared to processes that hog a CPU. We do so by bumping up the priority of I/O-bound processes and lowering the priority of CPU-bound processes based on their behavior. In addition to dynamically adjusting process priority to affect fairness, our scheduler needs to be efficient. We will adopt a dynamic scheduling framework based on a multilevel feedback queue used in operating systems such as UNIX Solaris.

---

2. Readings

1. Xinu setup from previous labs
2. Read Chapters 5-6 of the XINU textbook.

Please use a fresh copy of XINU, xinu-spring2024.tar.gz, but for preserving the greetings() function from lab1 and removing all code related to xsh from main() (i.e., you are starting with an empty main() before adding code to perform testing). As noted before, main() serves as an app for your own testing purposes. The TAs when evaluating your code will use their own main.c to evaluate your XINU kernel modifications.

---

3. Monitoring process CPU usage and response time [80 pts]

3.1 Clock interrupt and system timer

As discussed in class, XINU as well as Linux and Windows, can be viewed conceptually as being comprised of two main parts: upper half code that responds to system calls and lower half code that responds to interrupts. We introduced simple changes to XINU's upper half in Lab 1. Here we will consider a basic operation of the lower half involving clock interrupt handling. In the code forever.c (present in the course directory) even though forever.c does not contain any system calls, a process executing this code may still transition from user mode to kernel code due to interrupts while the process is running. For example, a hardware clock may generate an interrupt that needs to be handled by XINU's lower half. We noted that a process is given a time budget, called time slice or quantum, when it starts executing. Detecting that a process has depleted its time budget is achieved with the help of clock interrupts.

In computer architecture, the term interrupt vector is used to refer to an interface between hardware and software where if an interrupt occurs, it is routed to the responsible component of the kernel's lower half to handle it. In our case, the interrupt handling code of XINU that is tasked with responding to the clock interrupt is called the system timer. The clock interrupt on our x86 Galileo backends is configured to go off every 1 millisecond (msec), hence 1000 clock interrupts per second.

Our x86 backend processor supports 256 interrupts numbered 0, 1, ..., 255 of which the first 32, called faults (or exceptions), are reserved for dedicated purposes. The source of interrupts 0, 1, 2, ..., 31 are not external devices such as clocks and network interfaces (e.g., Bluetooth, Ethernet) but the very process currently executing on Galileo's x86 CPU. For example, the process may attempt to access a location in the main memory that it does not have access to which will likely result in interrupt number 13, called general protection fault (GPF). The process may attempt to execute an instruction that divides by 0 which maps to interrupt number 0. XINU chooses to configure clock interrupts at interrupt number 32 which is the first interrupt number that is not reserved for faults.

When a clock interrupt is generated, it is routed as interrupt number 32 with the help of x86's interrupt vector -- called IDT (interrupt descriptor table) -- to XINU's kernel code clkdisp.S in system/. Code in clkdisp.S, in turn, calls clkhandler() in system/clkhandler.c to carry out relevant tasks. Declare a new global variable, uint32 clkcounterms, in clkinit.c and initialize it to 0 (take a look at how other global variables, such as pid32 currpid, are declared and initialized). Modify clkdisp.S so that clkcounterms is incremented when clkdisp.S executes every 1 msec due to clock interrupt. Note that the software clock, clkcounterms, does not get updated if XINU's hardware clock that acts as the system timer is disabled. The more XINU disables clock interrupts the more inaccurate clkcounterms becomes as a timer that counts how many milliseconds have elapsed since a backend was bootloaded. Legacy XINU uses a global variable, uint32 clktime, that is updated in clkhandler() to monitor how many seconds have

elapsed since a backend was bootloaded. Test and assess whether both clkcounterms and clktime keep the same time using test code in main().

Note: Testing and verification of correctness of real-world software is rarely perfect. What we aim for is achieving a high standard which develops through practice.

3.2 CPU usage

We will modify XINU so that it can monitor the CPU usage of processes. For a process that has not terminated, its CPU usage will be the time (in unit of msec) that it has spent executing on Galileo's x86 CPU, i.e., in state PR_CURR.

Introduce a new process table field, uint32 prcpu, where XINU will record accumulative CPU usage of a given process (in unit of msec). For a process that is not in state PR_CURR, prcpu will contain XINU's estimate of its CPU usage from the time of its creation until now, i.e., the moment prcpu is inspected. For the current process, CPU usage will be the sum of prcpu and currcpu, where currcpu is a global variable of type uint32 that estimates the time (in msec) that the current process has spent in state PR_CURR after being context-switched in. Modify XINU so that when a context switch occurs, the prcpu field of the process being context switched out is updated as prcpu + currcpu. Then currcpu is reset to 0 so that the time that the new process (being context-switched in) spends in state PR_CURR can be tracked. Since XINU's system timer is set to interrupt every 1 msec increment currcpu in clkhandler().

Note: Please use the "volatile" qualifier when declaring currcpu.

3.3 Response time

Another important metric, especially for I/O-bound processes, is response time (in unit of msec) defined as the time a process resided in XINU's readylist before it became current. In other words, how long a ready process waited in line before it was serviced. Introduce two new process table fields, uint32 prresptime, and uint32 prctxswcount, where prctxswcount initialized to 0 when a process is created counts how many times a process was context-switched in, i.e., entered state PR_CURR from state PR_READY. The field prresptime initialized to 0 upon process creation sums the total time a process has spent since its creation in the readylist. We will estimate the average response time of a process over its lifetime by dividing prresptime by prctxscount. Introduce a new system call, syscall responsetime(pid32), in system/responsetime.c that returns the average response time rounded to an integer (in unit of msec) by dividing prresptime by prctxswcount.

When a process enters XINU's readylist, record the current time since bootloading a backend machine in a new process table field, uint32 prbeginready (in unit of msec). Reuse the global variable clkcounterms to keep track of system time elapsed since bootload. When a process transitions from PR_READY to PR_CURR, subtract prbeginready from clkcounterms. If the difference is 0, set the value to 1 (msec), in which case we overestimate response time. Now, add this difference to the prresptime. Also, increment prctxswcount. When coding

responsetime() we need to consider border cases. For example, a process may have become ready for the first time but not current. If so, prctxswcount will be 0 and let responsetime() return clkcounterms - prbeginready. If pid specified in the argument of responsetime() is that of a ready process (i.e., resides in readylist) then responsetime() will add clkcounterms - prbeginready to prresptime (but not update prresptime) and divide the resultant value by prctxswcount + 1, which is returned as an integer. Note that responsetime() returns values that estimate response time but it does not update prresptime and prctxswcount.

When modifying kernel code to implement CPU usage and response time monitoring, please check that your methods work correctly when a process that is context-switched in is a new process that is running for the first time. Keep in mind that for a new process ctxsw() does not return to resched() but directly jumps to user code (i.e., function pointer passed as the first argument of create()). Test your implementation to assess correctness.

Note: When implementing and testing code to monitor CPU usage and response time, use the legacy fixed priority XINU kernel, not the kernel with dynamic priority scheduling in Problem 4. Only after verifying that CPU usage and response time monitoring works correctly under fixed priority scheduling utilize it in Problem 4 for evaluating dynamic priority scheduling.

---

4. Dynamic priority scheduling [180 pts]

We will use a dynamic process scheduling framework based on multilevel feedback queues to adaptively modify the priority and time slice of processes as a function of their observed behavior.

4.1 Process classification: CPU-bound vs. I/O-bound

Classification of processes based on observation of recent run-time behavior must be done efficiently to keep the scheduler's overhead to a minimum. A simple strategy is to classify a process based on its most recent scheduling related behavior: (a) if a process depleted its time slice the process is viewed as CPU-bound; (b) if a process hasn't depleted its time slice and voluntarily gives up the CPU by making blocking call we will consider it I/O-bound. A third case (c) is a process that is preempted by a higher or equal priority process that was blocked but has become ready. When a process of equal priority becomes ready we have a choice of preempting or not preempting the current process. For simplicity, we will choose the former. We will reserve judgment and remain neutral by neither increasing nor decreasing its priority. Before resched() is called by kernel functions in the upper and lower half, XINU needs to make note of which case applies to the current process which, in turn, affects its future priority and time slice. In case (a), the current process is demoted in the sense that its priority is decreased following XINU's new scheduling table that follows the structure of the UNIX Solaris dispatch table. Demotion of priority is accompanied by an increase in time slice in accordance with the needs of a CPU-bound process. In case (b), the current process is promoted in the sense that its priority

is increased per XINU's scheduling table which also decreases its time slice. In case (c), no changes to the process's priority or time slice are made. We will remember the process's remaining time slice so that when it becomes current again its quantum is set to the saved time slice value.

To keep coding to a minimum, we will use sleepms() as a representative blocking system call for all other blocking system calls. Hence a process is considered I/O-bound if it makes frequent sleepms() system calls, thereby voluntarily relinquishing Galileo's CPU before its quantum expires.

For preemption events, we will only consider those triggered by system timer management in XINU's lower half, clkhandler(), which checks if a process in state PR_SLEEP needs to be woken up and inserted into the readylist. Time slice depletion is also detected by clkhandler() which calls resched() to let the scheduler determine who to run next.

4.2 XINU scheduling table

We will introduce a new kernel data structure, XINU scheduling table, which will implement a simplified version of the UNIX Solaris dispatch table containing 11 entries instead of 60. Hence the range of valid priority values is 0, 1, ..., 10. We will reserve priority value 0 for the idle/NULL process which must have priority strictly less than all other processes so that it only runs when there are no other ready processes in readylist. Hence normal processes take on priority values in the range 1, 2, ..., 10. All processes spawned using create() are subject to priority promotion/demotion based on recent behavior. The idle process is treated as a special case and its priority and time slice stay constant.

Define a structure in new kernel header file include/dynsched.h

```
struct dynsched_tab {
  uint16  ts_tqexp;         // new priority: CPU-bound (time quantum expired)
  uint16  ts_slpret;        // new priority: I/O-bound (called sleepms())
  uint16  ts_quantum;       // time slice associated with priority level
};
```

Declare, struct dynsched_tab dynprio[11], as global in initialize.c which is indexed by process priority and initialized so that for the idle process whose initial priority is set to 0 by sysinit() in system/initialize.c

```
dynprio[0].ts_tqexp = 0;
dynprio[0].ts_slpret = 0;
dynprio[0].ts_quantum = QUANTUM;
```

That is, the idle process whose priority is initialized 0 will not be promoted/demoted.

Modify create so that create() initializes a process's priority to 6 and time slice to 50 (msec). Hence a newly created process is assigned priority in the mid-range.

For the remaining entries of dynprio[i] for 0 < i < 11, configure their values as

dynprio[i].ts_tqexp = max{1, i-1}
dynprio[i].ts_slpret = min{10, i+1}
dynprio[i].ts_quantum = 110 - 10*i

A CPU-bound process gets demoted by decrementing its priority subject to there being a floor at priority level 1. An I/O-bound process gets promoted by incrementing its priority subject to there being a ceiling at priority level 10. The time slice values associated with priority levels 1, 2, 3, ..., 10 are 100, 90, 80, ..., 10 in accordance with the heuristic of assigning CPU-bound processes larger time budgets. If a process repeatedly depletes its time slice it will eventually hit rock bottom at priority level 1. Conversely, if a process repeatedly makes blocking system calls using sleepms() its processes reach the highest priority level 10.

4.3 Use of priority list in place of multilevel feedback queue

To simplify the coding of this lab, we will use the default XINU's readylist as if it were a multilevel feedback queue. The following explains how a multilevel feedback queue with 11 entries would be implemented if we were to replace the XINU's readylist. The queue is defined as a 1-D array of 11 elements of type struct whose fields contain pointers to the head and tail of a FIFO list. The FIFO list contains all ready processes at a specific priority level. The FIFO list at priority level i = 0, 1, ..., 10 is empty if the head and tail pointers are NULL. Insertion incurs constant overhead since a process is added at the end of the list pointed to by the tail pointer. Extraction incurs constant overhead since after determining the highest priority level at which there exists a ready process (i.e., FIFO is not empty), the head pointer is used to extract the first process in the list.

4.4 Testing and Performance Evaluation

Perform basic debugging by checking the internal operation of the modified kernel to verify the correct operation.

Benchmark apps
CPU-bound app*:* Code a function, void cpubnd(void), in system/cpubnd.c, that implements a while-loop. Define a new macro STOPPINGTIME set to 8000 in process.h. The while-loop checks if clkcounterms exceeds the threshold STOPPINGTIME set to 8000 (msec) in process.h. A process executing cpubnd() will terminate when clkcounterms has reached about 8 seconds. By design, a process executing cpubnd() hogs Galileo's CPU and is therefore an extreme case of a CPU-bound app.

I/O-bound app: Code a function, void iobnd(void), in system/iobnd.c, that implements a while-loop to check if clkcounterms has exceeded STOPPINGTIME. If so, iobnd() terminates. Unlike the body of cpubnd()'s while-loop which is empty, iobnd()'s while-loop body has a for-loop followed by a call to sleepms() with argument 100 (msec). Try different values for the bound of the inner for-loop such that it consumes several milliseconds of CPU time. It should not exceed

10 msec but otherwise is not important. The inner for-loop can contain arithmetic operations (even a nested for-loop) to help consume CPU time not exceeding 10 msec. Inspect the value of clkcounterms before and after the for-loop to calibrate the bound.

Benchmark output. Before terminating, cpubnd() and iobnd() print PID, "CPU-bound" or "I/O-bound", clkcounterms, CPU usage, and response time.

Workload scenarios We will consider homogenous and mixed workloads in benchmarks A-D.

Benchmark A. Spawn a workload generation process using create() that runs main() with priority 10. The process running main() spawns 6 app processes each executing cpubnd(). Call resume() to ready the 6 processes after creating them. The workload generation process terminates after creating and resuming the six benchmark processes. Output by the 6 app process upon termination at around 8 seconds of wall time should indicate approximately equal sharing of CPU time and similar response times.

Benchmark B. Repeat benchmark scenario A with the difference that the 6 app processes execute iobnd(). Since the apps are homogenous, their CPU usage and response time should be similar.

Benchmark C. Let the workload generator main() create 6 app processes, half of them executing cpubnd(), the other half iobnd().

Benchmark D. In the previous benchmarks all test processes were created at about the same time which serves a useful purpose of evaluating scheduling performance. In practice, processes are created over time which can inject significant complications for fair scheduling. For example, a process that was created 15 minutes in the past and has accumulated significant CPU usage will not be able to compete with newly created processes that have zero CPU usage. Until newly created processes "catch up" with the old process and rack up significant CPU usage, the old process will be starved which is unacceptable in operating systems. For this and other reasons, Linux CFS has to introduce ad hoc mechanisms that diminish the simplicity and elegance of fair scheduling.

In benchmark D, make the process running main() spawn two CPU-bound child processes, then sleep for 3 seconds. After waking up, make the process spawn two additional CPU-bound processes. To do so, the process's priority running main() may need to be set to 12 so that it is guaranteed to have the highest priority when it wakes up. Perform a thought experiment, i.e., simulate in your mind what approximate fraction of CPU usage you would expect the first two processes to receive, and similarly for the second group of two CPU-bound processes. Compare the observed output against your predicted results.

Bonus problem [25 pts]

Code an app, void joker(void), in system/joker.c that tries to exploit weaknesses in how our dynamic priority XINU scheduler classifies processes to promote/demote priorities to get the best of both worlds: large CPU usage and small response time compared to processes running cpubnd() and iobnd(). Benchmark using 5 processes where 2 run cpubnd(), 2 iobnd(), and 1 joker().

Note: The bonus problem provides an opportunity to earn extra credits that count toward the lab component of the course. It is purely optional.

Turn-in instructions

General instructions:

When implementing code in the labs, please maintain separate versions/copies of code so that mistakes such as unintentional overwriting or deletion of code is prevented. This is in addition to the efficiency that such organization provides. You may use any number of version control systems such as GIT and RCS. Please make sure that your code is protected from public access. For example, when using GIT, use git that manages code locally instead of its on-line counterpart github. If you prefer not to use version control tools, you may just use manual copy to keep track of different versions required for development and testing. More vigilance and discipline may be required when doing so.

The TAs, when evaluating your code, will use their own test code (replacing your main.c and main()) to drive your XINU code. The code you put inside main() is for your own testing and will, in general, not be considered during evaluation.

Specific instructions:

1. (Optional) The following is for those who want to systematically turn on/off the print statements that are either meant for debugging or those that are part of the lab (e.g., outputs from greetings()). Format for submitting written lab answers and kprintf() added for testing and debugging purposes in kernel code:

- For problems where you are asked to print values using kprintf(), use conditional compilation (C preprocessor directives #define combined with #ifdef and #endif) with macro XINUTEST (define the macro in include/process.h) to effect print/no print depending on if XINUTEST is defined or not. For your debug statements, do the same with macro XINUDEBUG.
- For example, given one line: kprintf(...) that is meant for debugging, write as the following instead:

```
#ifdef XINUDEBUG

        kprintf(...)

#endif
```

2. Electronic turn-in instructions:

i) Please make sure to follow the file and function naming convention as mentioned in the lab handout.

ii) Go to the xinu-spring2024/compile directory and run "make clean".

iii) Go to the directory where lab2 (containing xinu-spring2024/) is a subdirectory.

For example, if /homes/alice/cs354/lab2/xinu-spring2024 is your directory structure, go to /homes/alice/cs354

iv) Type the following command

```
turnin -c cs354 -p lab2 lab2
```

You can check/list the submitted files using

```
turnin -c cs354 -p lab2 -v
```

Please make sure to disable all debugging output before submitting your code.