CS 503 Fall 2022

Lab 3: Dynamic Priority TS Scheduling and Process Behavior (280 pts)

Due: 10/19/2022 (Wed.), 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 allocation of CPU cycles to processes. Through monitoring of process behavior we aim to compensate I/O-bound processes that voluntarily relinquish CPU with unspent quantum remaining by improving their responsiveness. That is, from the time an I/O-bound process that blocked becomes ready until when it becomes current is reduced in comparison to CPU-bound processes. The resultant resource consumption profile where CPU-bound processes beget more CPU cycles but incur longer response time relative to I/O-bound processes is the kernel's interpretation of "fairness." In addition to dynamically adjusting process priority to affect fairness, our scheduler needs to be efficient. We will utilize a data structure, called multilevel feedback queue, used UNIX Solaris which is an industrial grade OS to achieve constant overhead scheduling independent of the number of processes in the system. Lastly, TS scheduling employed by commodity operating systems implement heuristic methods that do not guarantee that CPU-bound processes will not starve in an operating environment of over-abundance of I/O-bound processes. We will implement a method for starvation prevention.

---

2. Readings

1. XINU set-up
2. Read Chapters 6-8 of the XINU textbook.

Please use a fresh copy of XINU, xinu-fall2022.tar.gz, but for preserving the myhello() function from lab1 and removing all code related to xsh from main() during testing. main() serves as an app for your own testing purposes. The TAs will use their own main() to evaluate your XINU kernel modifications.

---

3. Monitoring process resource consumption, performance, and behavior [80 pts]

An operating must monitor resource consumption, performance, and dynamic behavior of process in order to make scheduling decisions that achieve a desired system objective. We will add such measuring capability to XINU, reusing primitives from lab1 and lab2 where appropriate.

3.1 CPU usage

In lab2, we introduced process table field, uint32 prusercpu, wherein the time a process spent executing user code (in unit of msec) was estimated. We will reuse this feature in lab3 by porting the update code from clkhandler(). We will measure the time spent by a process in state PR_CURR which, in general, will include time spent executing kernel code when a kernel uses context-borrowing to run kernel code. In legacy XINU where both upper half and lower half are nonpreemptible, using clkhandler() in the lower half to track time spent executing kernel code will not work. Instead, we will use a 64-bit x86 counter called TSC (time stamp counter) which is driven by the CPU's clock rate to monitor the time a process spends in PR_CURR state. We will introduce a process table field, uint32 prtotalcpu that is initialized to 0. A new global variable, uint64

currstart, defined in process.h, will record the time in unit of processor tick (not our 1 msec system timer tick) when the current process changes its state from PR_READY to PR_CURR. Call getticks() and store the return value in currstart. We will normalize processor tick so that time spent in PR_CURR state can be converted to microsecond unit.

The function getticks() (see system/getticks.c) executes assembly instruction rdtsc using inline assembly and returns the value of x86's TSC counter (type uint64). When a process is context-switched out, call getticks() again and store its value in another global variable, uint64 currstop. Update prtotalcpu by adding (currstop - currstart) / 389. For our testing purposes, uint64 values can be converted to uint32 before performing arithmetic operations. The constant 389 is an empirical parameter for our x86 backend machines that converts time in unit of processor tick into microsecond unit. Accuracy of conversion is approximate which will suffice for benchmarking needs in Problem 4. The difference between prtotalcpu (in microsecond unit) and prusercpu (in msec) estimates the time a process spent executing kernel code. Introduce a new system call, syscall totcpu(pid32), in system/totcpu.c that returns total CPU usage in unit of msec (divide prtotalcpu by 1000) of the process specified in the argument, syserr if the argument is invalid.

3.2 Frequency of CPU usage

Monitor the number of times a process has been context-switched in to execute on a CPU by utilizing a new process table field, unsigned short prcurrcount. If update to currcount is at a different location in XINU code (i.e., function or assembly code block) from 3.1 indicate so in lab3.pdf.

3.3 Response time

We will measure response time of a process by noting the timestamp when a process enters into state PR_READY and subtracting it from the timestamp when its state changes to PR_CURR. To record the former, add process table field, uint64 prreadystart, which is updated using getticks(). When a process becomes current, add the difference (currstart - prreadystart) to new process table field, uint64 prtotalresponse, which is initialized to 0. Introduce a system call, syscall resptime(pid32), that returns the average response time of a process by dividing prtotalresponse (after converting to uint32) by prcurrcount. Make sure syserr is returned under relevant conditions so that resptime() doesn't cause the kernel to misbehave. resptime() returns response time in unit of millisecond, truncating digits at microsecond granularity and below. Monitor maximum response time in process table entry, uint32 prmaxresponse (in unit of msec), which will be used along with average response time as an indicator of starvation.

3.4 Preemption frequency

Performance metrics that are relevant to app programmers are exported via system calls. Other performance metrics may be primarily for internal kernel use and are hidden. Some may be accessible to knowledgeable programmers through system files that are left readable. An additional performance metric is the number of times a process has been preempted during its lifetime. We will consider two types of preemption: (i) the current process depletes its time slice and is context-switched out because of a ready process of equal or higher priority; (ii) an interrupt causes a blocked process of equal or higher priority to become ready and preempt the current process that has not depleted its time slice. In lab3, we will consider one external interrupt -- clock interrupt -- that readies processes waking up at the same time by calling wakeup(). If multiple sleeping processes wake up at the same time, resched() performs scheduling once after all of them have been readied using deferred scheduling conditional resched_cntl(). Keep track of type (i) preemption count in process table entry, unsigned short prpreemptcount1, and type (ii) count in, unsigned short prpreemptcount2. Note in lab3.pdf where you perform the updates and how you distinguish between the two preemption types.

Note: When implementing and testing code to monitor resource usage, it is recommended to do so using the legacy fixed-priority XINU kernel to remove potential bugs that your dynamic priority scheduler in Problem 4 may introduce. Then incorporate the monitoring code into XINU that implements dynamic priority scheduling.

---

4. Dynamic priority scheduling using multilevel feedback queue [200 pts]

We will implement a version of UNIX Solaris TS scheduling. The chief features are described in 4.1-4.4.

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 footprint to a minimum. As discussed in class, a simple strategy is to classify a process based on its most recent scheduling related behavior: (a) if a process depleted its time slice and needed to be preempted by the clock interrupt handler (preemption type (i) in 3.4) then the process is viewed as CPU-bound; (b) if a process hasn't depleted its time slice but is preempted by a higher priority process that became ready then the process's priority doesn't change; (c) if a process voluntarily relinquishes the CPU by making a system call (e.g., I/O related or sleep) that blocks then the process is viewed as I/O-bound. 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 values. In case (a), the current process is demoted in the sense that its priority is decreased following Solaris's dispatch table. Demotion of priority is accompanied by increase of time slice in accordance with the needs to a CPU-bound process. In case (b), the current process's status remains unchanged: priority does not change and the unspent time slice is remembered to be used when the process becomes current in the future. In case (c), the current process is promoted in the sense that its priority is increased per Solaris's dispatch table which decreases its time slice.

When case (a) applies, the current process must first be demoted before resched() selects which process to run next. When case (c) holds, the current process is promoted before it is blocked and context-switched out. For lab3, we will use sleepms() as a representative blocking system call for all other blocking system calls. Hence code changes to other blocking system calls are not needed. For preemption events we will only consider those triggered by system timer management in XINU's lower half, clkhandler(), which keeps track of time slice and handles processes that need to be woken up.

4.2 XINU dynamic scheduler dispatch table

We will implement a simplified dispatch table containing 10 entries instead of 60 in UNIX Solaris. Hence the range of valid priority values is 0, 1, ..., 9. We will reconfigure XINU so that the null/idle process is assigned priority value -1 upon creation during kernel initialization which the type pri16 allows. The status of the idle process never changes which preserves the system invariant that it only runs when there are no other ready processes in the system. All processes spawned using create() are viewed as falling in the TS scheduling class and subject to priority promotion/demotion based on recent behavior.

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

```
struct tsx_disp {
  unsigned short  tqexp;         // new priority: CPU-bound (time quantum expired)
  unsigned short  slpret;        // new priority: I/O-bound (sleep return)
  unsigned short  quantum;       // new time slice
};
```

Declare, struct tsx_disp dyndisp[10], in initialize.c which is initialized so that the i'th entry dyndisp[i] contains values

```
dyndisp[i].tqexp = max{0, i-1}
dyndisp[i].slpret = min{9, i+1}
dyndisp[i].quantum = 100 - 10*i
```

Hence a process that repeatedly depletes its time slice gets is priority decremented by 1 each time, eventually reaching rock bottom at 0. An I/O-bound process that repeatedly blocks gets its priority incremented by 1, reaching a cealing of 9. The lowest priority level 0 is associated with time slice 100 (msec), the highest priority level 9 with time slice 10 (msec). Code a modified version of create(), pid32 createtsx(), in system/createtsx.c that functions the same as create() but for its function prototype not having priority value

as third argument. The initial priority of a process spawned by createtsx() is assigned TSXINIT defined as 4 in dynsched.h.

4.3 Multilevel feedback queue

We will implement a multilevel feedback queue with 10 entries which replaces XINU's ready list to manage scheduling of ready processes. Define a multilevel feedback queue data structure in include/dynsched.h as

```
struct mfeedbqx {
  pid32 fifoqueue[NPROC];       // circular FIFO buffer of PIDs
  short head;                   // index of head of FIFO buffer
  short tail;                   // index of tail of FIFO buffer
  short count;                  // number of processes in the queue
};
```

Declare, struct mfeedbqx dynqueue[10], in initialize.c where dynqueue[i].fifoqueue[] is a circular FIFO buffer containing the PIDs of ready processes of priority i. dynqueue[i].count specifies the number of ready processes of priority i, dynqueue[i].head is an index of the head of the FIFO queue, dynqueue[i].tail is an index of the tail. Note that instead of using getmem() to dynamically manage FIFO linked lists at every priority level, we preallocate memory to reduce processing overhead. Initialize dynqueue[].count = dynqueue[].head = dynqueue[].tail = 0.

Code internal kernel function, short insertdynq(pri16, pid32), in system/insertdynq.c that inserts a process with PID specified by the second argument and priority specified by the first argument into the multilevel feedback queue. insertdynq() returns syserr upon error (insertion beyond capacity NPROC), and updated dynqueue[].count value upon success. Note that indices must be updated modulo NPROC since fifoqueue[] is a circular FIFO buffer. Code internal kernel function, pid32 extractdynq(void), in system/extractdynq.c that extracts and returns the PID of the ready process of highest priority from the multilevel feedback queue. If there are two or more ready processes of highest priority, the one at the front of the queue is returned. extractdynq() returns syserr if there are no ready processes. This implies that the idle/null process is the only ready process. Code internal kernel function, pri16 inspectmaxprio(void) in system/inspectmaxprio.c which returns the maximum priority of ready processes in the multilevel feedback queue. If the queue is empty, inspectmaxprio() returns syserr which is consistent with the priority assigned to idle/null process. Overhead of insertdynq() and extractdynq() do not depend on the number of ready processes, hence constant.

4.4 Starvation prevention

Dynamic priority scheduling, in general, does not guarantee that starvation will not occur. As a preventative measure, we may periodically check the maximum response time of ready processes and boost the priority of those whose max response time exceeds a threshold. Starvation prevention will be enabled by a flag STARVATIONPREVENT defined in system/process.h if it is set to 1. If set to 0, our scheduler will not perform starvation prevention. The inspection period is specified by STARVATIONPERIOD which will defined as 100 (msec) in process.h. The boost threshold is specified in process.h as STARVATIONTHRESHOLD 500 (msec). The amount by which the priority of a starving process will be increased is specified as PRIOBOOST which will be set as 3. We will implement starvation prevention by clkhandler() calling an internal kernel function, void preventstarvation(void), coded in system/preventstarvation.c every STARVATIONPERIOD. preventstarvation() iterates through the priority levels i in the multilevel feedback queue dynqueue[i], inspecting the waiting time of the first element (if not empty) at level i. Waiting time is calculated by subtracting prreadystart of the process from the value returned by getticks() (i.e., current time). If the process's waiting time exceeds STARVATIONTHRESHOLD, its priority is increased by PRIOBOOST (resultant priority is capped at 9). If the process's pre-boost priority was less than 9, it is dequeued from its old FIFO queue in dynqueue[i] and enqueued in its new FIFO queue in dynqueue[j] where j > i is the new priority value after applying boosting. Priority boosting is applied to the front element of FIFO queues only.

Test and verify that the components in 4.1-4.4 are working correctly.

4.5 Performance evaluation

Benchmark apps Evaluate the performance of your XINU implementation of dynamic priority scheduling using multilevel feedback queue using benchmark workloads. First, code a function, void cpubound(void), in system/cpubound.c, that implements a while-loop. The while-loop checks if vfineclkcounter exceeds a threshold which is defined by STOPTIME set to 10000 (msec) in process.h. A process executing cpubound() will terminate when vfineclkcounter has reached about 10 seconds which is the default benchmark duration. A process spawned by calling createtsx() that executes cpubound() will represent a CPU-bound process.

Second, code a function, void iobound(void), in system/iobound.c, that implements a while-loop to check if vfineclkcounter has exceeded STOPTIME. If so, iobound() terminates. Unlike the body of cpubound()'s while-loop which is empty, iobound()'s body has a for-loop followed by a call to sleepms() with argument 80 (msec). Calibrate the bound of the for-loop such that it takes about 2 msec of CPU time. Add code in the body of for-loop (including nested for-loops) to consume roughly 2 msec of CPU time.

Before terminating, cpubound() and iobound() print "cpubound" or "iobound", PID, total CPU usage, user CPU usage, average and maximum response time, number of context-switch in events, counts of type (i) and type (ii) preemption events.

Workload scenarios We consider homogenous and mixed workloads in benchmarks A-C. Benchmark D considers mixed workloads where starvation may occur.

Benchmark A. Spawn a workload generation process using create() that runs main() with priority 9. main() spawns using createtsx() 8 app processes each executing cpubound(). Call resume() to ready the 8 processes after creating them. Output produced by the 8 app process upon termination at around 10 seconds of wall time should indicate approximately fair sharing of CPU time. Discuss your finding in lab3.pdf.

Benchmark B. Repeat benchmark scenario A with the difference that the 8 app processes execute iobound(). Since the apps are homogenous, their resource consumption and performance profile should be similar. Discuss your finding in lab3.pdf which includes a comparison of their profiles compared to benchmark A.

Benchmark C. Let the workload generator main() create 8 app processes, half of them executing cpubound(), the other half iobound(). Analyze your finding in lab3.pdf.

Benchmark D. With the parameters of the workload above it is unlikely that starvation where one or more ready processes are not allocated CPU cycles for an extended time interval will occur. In our case, we artificially interpret "extended" as greater than STARVATIONTHRESHOLD (msec). Devise a workload where starvation of CPU-bound processes will occur. The general idea is to set up the number of I/O-bound processes and their workload parameters such that one or more CPU-bound processes will not get to run due to their lower priority values compared to I/O-bound processes that make blocking system calls. A sufficient benchmark scenario is multiple I/O-bound process vs. a single CPU-bound process. Perform a benchmark run with STARVATIONPREVENT set to 0. Monitor maximum and average response time for indication of starvation. Run the same test case with starvation prevention enabled and compare the resultant output. The parameter settings for starvation prevention STARVATIONTHRESHOLD, STARVATIONPERIOD, PRIOBOOST may be inadequate to effect noticeable starvation depending on your workload configuration. If so, make changes to the parameters, keeping in mind that efficiency is important when considering kernel modification. Discuss your results in lab3.pdf.

Bonus problem [25 pts]

The workloads considered in 4.5 are static in the sense that all processes are created at about the same time as a batch. In the real-world workloads are time-varying, i.e., processes are created at different times. Static workloads, nonetheless, are of practical relevance in that they can be used to capture load placed on a kernel during peak periods, in contrast to average load. Accurate generation of time-varying workloads may entail spawning of new process at specific times which requires the workload generation process to execute at designated times. A workload generation process that itself undergoes dynamic priority scheduling may not be able to generate workloads at designated times.

In commodity kernels such as Linux and Windows, real-time scheduling classes are defined whose processes do not undergo dynamic priority changes and their fixed priority exceeds processes in the TS class. Define a RT scheduling class that utilizes priority value 10. Extend the multilevel feedback queue so that it has 11 levels, struct mfeedbqx dynqueue[11]. TS scheduling works as before with priority ceiling set at 9. However, when resched() considers which process to run next RT processes in dynqueue[10] are considered first. Only if the FIFO queue at dynqueue[10] is empty are TS processes at priority level 0, 1, ..., 9 scheduled. To spawn a RT process, modify createtsx() to implement, pid32 createrтx(), in system/createrтx.c that reintroduces the third argument of create() as the fixed priority of XINU RT processes. In our version, only priority value 10 will be considered valid for RT. Rerun Benchmark A in 4.5 using createrтx() instead of create() to spawn the workload generation process executing main(). Verify that XINU with RT class works correctly.

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 (principally 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.

If you are unsure what you need to submit in what format, consult the TA notes link. If it doesn't answer your question, ask during PSOs and office hours which are scheduled M-F.

Specific instructions:

1. Format for submitting written lab answers and kprintf() added for testing and debugging purposes in kernel code:

- Provide your answers to the questions below in lab3.pdf and place the file in lab3/. You may use any document editing software but your final output must be exported and submitted as a pdf file.
- For problems where you are asked to print values using kprintf(), use conditional compilation (C preprocessor directives #define combined with #if and #endif) with macro XINUTEST (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.

2. Before submitting your work, make sure to double-check the TA Notes to ensure that any additional requirements and instructions have been followed.

3. Electronic turn-in instructions:

i) Go to the xinu-fall2022/compile directory and run "make clean".

ii) Go to the directory where lab3 (containing xinu-fall2022/ and lab3.pdf) is a subdirectory.

For example, if /homes/alice/cs503/lab3/xinu-fall2022 is your directory structure, go to /homes/alice/cs503

iii) Type the following command

turnin -c cs503 -p lab3 lab3

You can check/list the submitted files using

turnin -c cs503 -p lab3 -v

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

Back to the CS 503 web page