CS 354 Spring 2025

Lab 3: Monitoring Process Behavior and Dynamic Priority Scheduling [300 pts]

Due: 3/5/2025 (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 sharing of CPU cycles among processes. First, we add instrumentation code and new system calls so that total CPU usage, starvation, and average response time of app processes is monitored. Second, we 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 classifying processes as CPU- or I/O-bound based on recent behavior. This brings XINU closer in line with how scheduling is performed in commodity operating systems such as Linux and Windows.

---

2. Readings

1. XINU set-up
2. Read Chapters 5-6 of the XINU textbook.

Please use a fresh copy of XINU, xinu-spring2025.tar.gz, but for preserving the output from lab1 identifying authorship, 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 GTAs when evaluating your code will use their own main() to evaluate your XINU kernel modifications.

---

3. Monitoring process CPU usage and response time [100 pts]

3.1 Total CPU usage

We will modify XINU so that it monitors CPU usage of processes. For a process that has not terminated, its total 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 since creation. Introduce a new global variable, uint32 currcputime, declared in system/initialize.c that tracks CPU usage of the

current process (in msec unit) since it was context-switched in. The global currcputime is related to the value QUANTUM - preempt since XINU updates the global variable preempt in clkhandler() to monitor the current process's remaining time slice. However, the current process, upon depleting its time slice at which time XINU's scheduler resched() is called to determine which process to run next, may continue to run on the CPU if it has (strictly) highest priority. If so, preempt will be reset to QUANTUM to assign a fresh time slice but currcputime is not reset. Instead, it has to keep tallying CPU usage of the current process.

On the other hand, if the current process is context-switched out then a new process table field, uint32 prcputotusage (initialized to 0 upon process creation) is updated in include/process.h by adding currcputime to monitor total CPU usage. After the updating prcputotusage, currcputime is reset to 0 so that it can track CPU usage of the process being context-switched in. Instead of using the macro QUANTUM, declare a global variable, uint16 quantumts, in initialize.c (its default value set to QUANTUM) that will be used in Problem 4 when performing dynamic priority scheduling to assign the time slice of a process before it is context-switched in.

When implementing a solution, explicitly consider the case where a process being context-switched in is running for the first time after creation. Annotate your mods in XINU code to highlight where code changes were made and a brief indication of why. Test and verify that your XINU modification works correctly. Describe in lab3ans.pdf your method for assessing correctness and what test cases you considered.

3.2 System call getcputot()

Implement a new system call, int32 getcputot(pid32), in system/getcputot.c that returns total CPU usage of the process specified in the argument. If the PID belongs to a process that is not in state PR_CURR, getcputot() returns the value stored in process table field prcputotusage. If the PID specifies the current process, the return value is slightly different. Make sure to consider when getcputot() should return SYSERR. Test and verify that your implementation works correctly.

3.3 Starvation detection

Similar to UNIX Solaris, we would like to detect if a ready process has not received CPU cycles for RECENTWIN seconds, a macro to be defined in a new header file, include/xsh.h, with

default value 1. Modify XINU including clkhandler() the lower half so that every RECENTWIN seconds each ready process is inspected to determine if it is starving, i.e., has been waiting to run for RECENTWIN seconds or more. To do so, introduce new process table fields, uint32 prreadystart, which records when a process has become ready by entering XINU's readylist. Re-use the 1 msec resolution counter, unsigned int clkcounterfine, from Problem 4.1 in lab1 as XINU's 1 msec resolution wall clock time to set prreadystart. If the difference between clkcounterfine and a ready process's prreadystart exceeds 1000, a new process table field, uint16 prstarvation, is set to 1 (initial value 0). prstarvation is set to 0 if a ready process is selected to run. Introduce a process table field, uint16 prstarvecount, initialized to 0 that counts how many times a process has encountered starvation. Test and verify that your implementation works correctly. Describe in lab3ans.pdf your method for assessing correctness.

3.4 Average response time

An important metric, especially for I/O-bound processes, is response time (in unit of msec) defined as the time a process has resided in XINU's readylist before it became current. In 3.3

we used recent response time with threshold RECENTWIN as an indicator of CPU starvation. We will augment XINU's monitoring capability to estimate average response time experienced by a process which will be used in Problem 4 to assess scheduling performance that aims to reward I/O-bound processes that do not consume much CPU time with better response times. Introduce two new process table fields, uint32 prnumready, and uint32 prsumresponse, where prnumready counts the number of times as process has transitioned into state PR_READY. For brevity we will consider test scenarios where a process transitions into PR_READY from states PR_CURR, PR_SUSP, and PR_SLEEP. The process table field prsumresponse is the sum of the response times (i.e., time waiting in the readylist) over the prnumready times a process has entered XINU's readylist.

Implement a system call, uint32 meanresponsetime(pid32), in system/meanresponsetime.c that returns an integer (in unit of msec) obtained by rounding average response time prsumresponse / prnumready. As with all system calls newly implemented in the course, please perform careful sanity checks and return SYSERR if an abnormal condition is detected. For example, a process specified in the argument of meanresponsetime() may have been created but never put into

ready state. Test and verify that your code works correctly. Describe in lab3ans.pdf your method for performing application layer testing.

Note: When implementing and testing monitoring code, use the legacy fixed priority XINU kernel, not the kernel with dynamic priority scheduling in Problem 4. Only after verifying that XINU's monitoring features work correctly under fixed priority scheduling will they be utilized in Problem 4 for evaluating dynamic priority scheduling. Hence fixed priority scheduling is useful for debugging and testing monitoring code implemented in the XINU kernel.

---

4. Dynamic priority scheduling [200 pts]

We will use a dynamic process scheduling framework based on UNIX Solaris 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 relinquishes 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. We will choose the latter in recognition of the fact that the current process has time slice remaining.

Before resched() is called by kernel functions in the upper and lower half, XINU needs to make note of which of the three cases applies to the current process which determines 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 increase of time slice in accordance which comports with the needs of CPU-bound processes. In case (b), the current process is promoted in the sense that its priority is

increased as dictated by XINU's scheduling table. Its time slice is decreased. In case (c), no changes are made to the preempted process's priority and time slice. We will remember the process's remaining time slice so that when it becomes current again its time slice is set to the saved value. The process is not given special consideration for having been preempted. That is the hard reality of a XINU process's life under Solaris motivated scheduling.

To keep coding volume low, 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 time slice expires. Calling sleepms() with argument 0 results in the resched() being called which can be useful in some application environments. For simplicity, we will disallow sleepms() to be called with 0 sleeptime to prevent a process from being classified as I/O-bound and amplify its priority if no ready processes exist in the system. To prevent this potential loophole, modify sleepms() so that it returns SYSERR if called with argument 0.

For lower half XINU events that prompt calling resched() that may cause preemption of the current process, we will consider invocations from clkhandler() only for two reasons. First, clkhandler() detects that the current process has depleted its time slice. Second, one or more processes are woken up and readied by insertion into the ready queue. Until we discuss sleep events in detail later in the course under clock device management, we will treat XINU's sleep queue as a black box where sleeping processes are enqueued and waking processes are dequeued. The latter is achieved by calling wakeup() on the processes waking up (there may be more than one process needing to wake up at the same time) which inserts all of them into XINU's readylist and invoking resched()'s scheduling actions once after the process is complete to pick a highest priority process among multiple processes entering the readylist. This is one application of "deferred" scheduling.

4.2 XINU scheduling table

We will introduce a new kernel data structure, XINU scheduling table (XST), which will implement a simplified version of the UNIX Solaris dispatch table containing 8 entries instead of 60. Hence the range of valid priority values is 0, 1, ..., 7. 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. Normal processes take on priority values in the range 1, 2, ..., 7. All processes spawned using create() are subject to priority promotion/demotion based on recent behavior. The third argument of create() specifying priority is ignored. The idle process is treated as a special case where its priority and time slice stay constant.

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

```
struct xst {
  uint16  xstslpret;        // new priority for I/O-bound processes
  uint16  xstqexp;         // new priority for CPU-bound processes
  uint16  xstquantum;      // time slice associated with a priority
level
};
```

Declare, struct xst xstds[8], as global in initialize.c. Its entries are indexed by process priority. xstds[] is initialized so that for the idle process whose initial priority is set to 0 in sysinit() in system/initialize.c

```
xstds[0].xstslpret = 0;
xstds[0].xstqexp = 0;
xstds[0].xstquantum = QUANTUM;
```

Hence the idle process whose priority is initialized 0 will not be promoted/demoted. Set the macro QUANTUM to 10.

For normal processes spawned by calling create(), modify create() so that its priority is set to 4 and time slice to 40 (msec). A newly created process is assigned priority mid-range in the absolute scale 1, 2, ..., 7 although its relative -- and effective -- priority will depend on the priority of other ready processes and the current process. If they are aggregate at the lower end then priority value 4 will imply high priority, the opposite if they are aggregate at the upper end. If dynamic priorities are evenly distributed 4 will indeed mean mid-range.

For the remaining entries of xstds[i] for i > 0, configure their values as

```
xstds[i].xstslpret = min{7, i+1}
xstds[i].xstqexp = max{1, i-1}
xstds[i].xstquantum = 80 - 10*i
```

Hence a newly created process with initial priority 4 will be assigned a time slice of 40 msec. A CPU-bound process with current priority i gets demoted by decrementing its new priority to xstds[i].xstqexp. An I/O-bound process gets promoted by incrementing its priority to xstds[i].xstslpret. As is the case for the Unix Solaris TS scheduler, time slice decreases as priority increases. 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 reaches highest priority level 7.

4.3 Use of priority list in place of multilevel feedback queue

It is straightforward to implement a multilevel feedback queue with 8 entries which replaces XINU's readylist by defining a 1-D array of 8 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, ..., 7 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.

Since we will not be evaluating kernel performance using hundreds of processes at which point the advantage of constant overhead may become significant, we will continue to utilize XINU's readylist to manage ready processes which also reduces coding effort. Our focus in Problem 4 is dynamically adjusting the priorities of processes based on whether they

behave in a CPU- or I/O-bound manner, and using the instrumentation tools from Problem 3 to assess if equitable sharing of CPU cycles is being achieved. Coding a multilevel feedback queue data structure is an option we will not exercise since a priority queue emulates the behavior of a multilevel feedback queue but for incurring linear instead of constant scheduling overhead. For example, six ready processes P1, P2, ..., P6 with priorities 3, 5, 2, 5, 7, 1, and idle process P0 with priority 0 will form the priority list

-> (P5,7) -> (P2,5) -> (P4,5) -> (P1,3) -> (P3,2) -> (P6,1) -> (P0,0)

where P5 is at the head of the list. P2 and P4 of equal priority are next. If a sleeping process P21 of priority 5 became ready, XINU's insert() kernel function would place the P21 after P4 in the priority list

-> (P5,7) -> (P2,5) -> (P4,5) -> (P21,5) -> (P1,3) -> (P3,2) -> (P6,1) -> (P0,0)

A multilevel feedback queue and its counterpart priority list

perform an equivalent function. Overhead, however, is worlds apart.

4.4 Safety net for starvation prevention

Define a macro STARVATIONPREV in header file include/xsh.h which may be set to 1 to activate starvation prevention, 0 to disable it. clkhandler() has been modified in 3.3 to perform starvation detection every RECENTWIN second. Set RECENTWIN to default value 1. If a ready process's prstarvation field is marked 1 (i.e., has not received CPU cycles for at least RECENTWIN seconds) then change its priority to maximum value 7. To modify the priority of a process already in the readylist, code a kernel function, status modprioready(pid32, qid16, pri16), in system/modprioready.c that modifies XINU readylist so that the process specified in the first argument has its priority changed to the value specified in the third argument and placed in its proper place in the priority list. This can be done in multiple ways such as removing the specified process from readylist then using XINU's insert() function to insert the process back into readylist with its modified priority.

When modifying scheduling code it is important to consider concurrent events such as clkhandler() determining that the

current process has depleted its time slice (part of legacy XINU code) and detecting that a process has exceeded the threshold for starvation. If so, resched() should be invoked once to select which process to run next after priorities of all ready processes have been updated. This can be achieved in multiple ways, the specific implementation being up to you.

4.5 Testing and performance evaluation

Perform basic debugging by checking the internal operation of the modified kernel to verify correct operation. Use macro XINUDEBUG for this purpose which is then disabled to suppress output of debug messages in your submitted code. Use macro XINUTEST to enable output of the values during benchmarking where equitable sharing of CPU cycles gauged by total CPU usage and improved response time of I/O-bound processes is evaluated.

4.5.1 Benchmark apps: CPU- and I/O-bound

CPU-bound app. Code a function, void cpuproc(void), in system/cpuproc.c, that implements a while-loop. The while-loop checks if clkcounterfine exceeds a threshold which is defined by macro STOPTIME set to 10000 (msec) in xsh.h. A process

executing cpuproc() will terminate when clkcounterfine has reached about 10 seconds. By design, a process executing cpuproc() hogs Galileo's CPU and is therefore an extreme case of a CPU-bound app.

I/O-bound app. Code a function, void ioproc(void), in system/ioproc.c, that implements a while-loop to check if clkcounterfine has exceeded STOPTIME. If so, ioproc() terminates. Unlike the body of cpuproc()'s while-loop which is empty, ioproc()'s while-loop body has a for-loop followed by a call to sleepms() with argument 50 (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 clkcounterfine before and after the for-loop to calibrate the bound.

Benchmark output. Before terminating, cpuproc() and ioproc() should output PID, "cpuproc" or "ioproc", clkcounterfine, total CPU usage, average response time, and prstarvecount. Perform

a test with, and without, STARVATIONPREV enabled in the fourth workload (Benchmark D).

4.5.2 Workload scenarios

We will 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 99. The process running main() spawns 4 app processes each executing cpuproc(). Call resume() to ready the 4 processes after creating them. The workload generation process terminates after creating and resuming the four benchmark processes. For ease of benchmarking modify create() so that it checks the third argument. If its value is 99, set the initial priority of the workload generation process to 7. This will assign artificially elevated priority so that the parent is able to generate all child processes without being preempted. Output by the 4 app process upon termination at around 10 seconds of wall time should indicate approximately equal sharing of CPU time and similar average response times. Given that there are only 4

processes and time slice in XST is sub-70 msec starvation should not occur. Discuss your results in lab3ans.pdf.

Benchmark B. Repeat benchmark scenario A with the difference that the 4 app processes execute ioproc(). Since the apps are homogenous, their CPU usage and average response times should be similar. Starvation should not be a factor. Compare the results of benchmarks A and B. Discuss your finding in lab3ans.pdf.

Benchmark C. Let the workload generator main() create 4 app processes, half of them executing cpuproc(), the other half ioproc(). Discuss your results in lab3ans.pdf.

Benchmark D. Devise a workload of many I/O-bound processes and a few CPU-bound processes such that, collectively, the I/O-bound processes will hog the CPU and cause starvation of CPU-bound processes. Code your skewed hybrid workload generator, void shybrid(void), in system/shybrid.c. Perform benchmark tests with STARVATIONPREV disabled and enabled. Evaluate if the starvation prevention mechanism of 4.4 is effective. Discuss your finding in lab3ans.pdf.

Note: Please annotate changes to XINU code. For code in new files provide descriptive comments on what the key parts are doing. The top of the file should state who is the author of the code, semester and year. For modifications to legacy code, prepend your comments with "MOD" and your initial followed by colon. For example, "// MOD AS: ..." or "/ MOD AS: ... */" for Alice Smith. Annotation need not be extensive, serving to help a competent coder follow what your code is doing. Submissions that are lacking will incur a 20 point reduction.*

Bonus problem [30 pts]

For Benchmark A, create a fifth app process executing your code, void rogue(void), in system/rogue.c, that exploits potential vulnerabilities of the dynamic priority scheduler so that it gets a noticeably larger share of CPU cycles than the four processes running cpuproc(). On the surface this seems to go against TS scheduling where to get a larger CPU share, higher priority is needed, and to get higher priority, not using much CPU time through blocking system calls is needed. Describe your approach to implementing rogue() and discuss your results in lab3ans.pdf.

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 lab3ans.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 #ifdef 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-spring2025/compile directory and run "make clean".

ii) Go to the directory where lab3 (containing xinu-spring2025/ and lab3ans.pdf) is a subdirectory.

For example, if /homes/alice/cs354/lab3/xinu-spring2025 is your directory structure, go to /homes/alice/cs354

iii) Type the following command

turnin -c cs354 -p lab3 lab3

You can check/list the submitted files using

turnin -c cs354 -p lab3 -v

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

Back to the CS 354 web page