CS 354 Fall 2023

Lab 3: Monitoring Process Behavior and Dynamic Priority Scheduling [260 pts]

Due: 10/18/2023 (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 instrumention code and new system calls so that CPU usage and 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-fall2023.tar.gz, but for preserving the helloworld() 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 GTAs when evaluating your code will use their own main() to evaluate your XINU kernel modifications.

---

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

3.1 CPU usage

We will modify XINU so that it can monitor 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. Estimating CPU usage accurately requires understanding device management (of which hardware clocks are a part) which we will cover later in the course. We will implement a method that is adequate for our purpose of gauging whether our dynamic priority scheduler implemented in Problem 4 achieves fair allocation of CPU cycles to processes.

Introduce a new process table field, uint32 prtotcpu, where XINU will record CPU usage (in unit of msec). For a process that is not in state PR_CURR, prtotcpu will contain XINU's estimate of its total CPU usage from the time of its creation until now, i.e., the moment prtotcpu is inspected. For the current process, CPU usage will be the sum of prtotcpu and cputime where cputime is a global variable of type uint32 to be declared in system/initialize.c 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 old process's prtotcpu field is updated as prtotcpu + cputime. Note that "old process" refers to the process being context-switched out. If cputime equals 0, set it to 1 before calculating prtotcpu + cputime. For processes that consumed less than 1 msec of user CPU time before context-switching out, we round up CPU usage to 1 msec resulting in overestimation. Reset cputime to 0 so that the time that the new process spends in state PR_CURR can be monitored.

Since XINU's sytem timer is set to interrupt every 1 msec, increment cputime in clkdisp.S similar to how clkcountermsec was updated in Problem 4.1 of lab1. Since XINU disables external interrupts when executing upper and lower half kernel code, cputime underestimates total CPU usage when XINU executes kernel code. This corresponds to user CPU time in Linux that tracks CPU usage of a process in user mode.

Introduce a new system call, syscall totcpu(pid32), in system/totcpu.c that returns total CPU usage in unit of msec of the process specified in the argument. If the argument specifies the PID of the current process, totcpu() returns prtotcpu + cputime. prtotcpu of the current process is updated only when it is context-switched out. If the argument specifies a process that is not in state PR_CURR, return prtotcpu. Check the PID argument of totcpu() to verify that it is valid, and return SYSERR if not. Test your implementation to gauge correctness. Describe in lab3.pdf your method for doing so. Given that cputime is incremented by assembly code in clkdisp.S, is it necessary to use the "volatile" qualifier when declaring the global variable in initialize.c? Discuss your reasoning.

3.2 Response time

An 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. Introduce two new process table fields, uint32 prtotresp, and uint32 prtotready, where prtotready initialized to 0 when a process is created counts how many times a process transitioned to state PR_READY. When updating prtotready only consider XINU process states that we have covered thus far in the course. For example, PR_RECV need not be considered since haven't discussed sending and receiving messages. The field prtotresp initialized to 0 upon process creation sums the total time a process has spent since its creation waiting in the readylist. We will estimate average response time of a process over its lifetime by dividing prtotresp by prtotready. Introduce a new system call, syscall avgresponse(pid32), in system/avgresponse.c that returns average response time rounded up to an integer (in unit of msec) by dividing prtotresp by prtotready.

When a process enters XINU's readylist, record the current time since bootloading a backend machine in a new process table field, uint32 prreadystart. Reuse the global variable, uint32 clkcountermsec, from Problem 4.1, lab1, to keep track of system time elapsed since bootload. When a process transitions from PR_READY to PR_CURR, subtract prreadystart from clkcountermsec. If the difference is 0, set the value to 1 (msec), which overestimates response time.

Test your implementation to assess correctness. Describe in lab3.pdf your method for doing so. In the case where a newly created process becomes ready for the first time, discuss if the above method for calculating response time incurs inaccuracies. There is no need to rectify any inaccuracies.

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.

3.1 CPU Usage

- CPU usage = Time spent executing (i.e. prstate == PR_CURR) while process is still not terminated

1. New process table field uint32 prtotcpu to record CPU usage in msec ✓
   - ↳ If not PR_CURR, prtotcpu = Moment prtotcpu is inspected - time of creation
   - ↳ Else, prtotcpu = prtotcpu + cputime
2. Global variable uint32 cputime in system/initialize.c to estimate how many msec the current process has spent in PR_CURR after being context-switched in ✓
3. When context-switch happens, old process' prtotcpu = prtotcpu + cputime ✓
   - ↳ If cputime == 0, set it to 1 ✓

   Only updated when process is context-switched out
4. Reset cputime to 0 for the new process ✓
5. Increment cputime in clkdisp.S similar to lab1 ✓
6. Write syscall totcpu(pid32) in system/totcpu.c that returns total CPU usage (msec)
   - ↳ If PID is in PR_CURR, return prtotcpu + cputime ✓
   - ↳ Else, return prtotcpu ✓
   - ↳ If PID is not valid, return SYSERR ✓

3.2 Response Time

- Response time: The time a process resides in XINU's readylist before it becomes current

1. Process table field uint32 prtotready counts # of times it changes to PR_READY
   - ↳ Initialized to 0 when created ✓
   - ↳ Update for process states we have covered (exclude PR_RECV) ? → ready.c, resched.c
2. Process table field uint32 prtotresp sums total time waiting in readylist
   - ↳ Initialized to 0 when created ✓

   How to update?
3. syscall avgresponse(pid32) in system/avgresponse.c
   - ↳ Returns prtotresp / prtotready rounded up to an integer ✓
4. Process table field uint32 preadystart records current time since bootloading a backend machine when the process enters XINU's readylist
5. Reuse uint32 clkcountermsec from lab1 to keep track of time since bootload ✓
   - ↳ When process changes PR_READY → PR_CURR, clkcountermsec - preadystart
   - ↳ If difference is 0, set it to 1

   resched.c

   store in prtotresp?

4. Dynamic priority scheduling [180 pts]

We will use a dynamic process scheduling framework based on process behavior 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 preempt the current process. We will remain neutral and not change the priority of the preempted process.

We will repurpose the process table field, pri16 prprio, to indicate both a priority value and the classification of a process as CPU- or I/O-bound. The former preserves backward compatibility. In the case of (a), prprio is set to 1 which classifies a process as CPU-bound. In the case (b), prprio is set to 2 which denotes a process as I/O-bound. In the case of (c), the previous value of prprio is maintained. We will set prprio to 0 to indicate that the process is XINU's idle/null process. The idle process is handled as a special case so that it only runs when there are no current or ready processes in the system. Setting prprio to 0 which is strictly less than CPU- and I/O-bound processes achieves that. Define three macros, #define QUANTUMIO 5, #define QUANTUMCPU 50, #define QUANTUMIDLE 100, in include/process.h that specify the time slice assigned to I/O-bound, CPU-bound processes, and the idle process, respectively.

4.2 Upper half blocking and lower half preemption

A process is deemed I/O-bound when it makes a blocking system call in the upper half of the kernel. The current process may be preempted by another process of equal or higher priority by the actions of the kernel's lower half which responds to interrupts. To keep coding to a minimum, we will use sleepms() as a representative blocking system call for all other blocking system calls. Hence in benchmark testing a process is considered I/O-bound if it makes frequent sleepms() system calls, thereby voluntarily relinquishing Galileo's CPU before its quantum expires. If a process calls sleepms() with an argument that is negative, sleepms() will return SYSERR. An argument value of 0 is allowed in XINU which prompts the kernel to call resched(). We will treat this case as I/O-bound behavior since an I/O-bound ready process would preempt the current process.

For preemption events we will only consider those triggered by system timer management in XINU's lower half, clkhandler(). The first scenario we will consider is checking if a process in state PR_SLEEP needs to be woken up and inserted into the readylist. The second scenario is time slice depletion where clkhandler() calls resched() to let the scheduler determine who to run next. Note the classification in 4.1 must be implemented by modifying relevant upper and lower half kernel code.

Modify create() so that it ignores the third argument and sets the priority of newly created processes to 1. In benchmark testing we will use the process executing main() to generate the workload processes and then terminate. Assign to this process priority value 3 by calling chprio() from main(). Since the process executing main() terminates after creating workload processes described in 4.3 it will not affect overall performance evaluation. The global variable preempt used to track the current process's remaining time slice must be set to the appropriate value for CPU-bound and I/O-bound processes, as well as for the idle process, in different parts of kernel code.

4.3 Testing and performance evaluation

Perform basic debugging by checking the internal operation of the modified kernel to verify correct operation. Utilize 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 CPU usage and improved response time of I/O-bound processes is evaluated.

Benchmark apps *CPU-bound app*. Code a function, void cpuproc(void), in system/cpuproc.c, that implements a while-loop. The while-loop checks if clkcountermsec exceeds a threshold which is defined by macro STOPCOND set to 10000 (msec) in system/initialize.c. A process executing cpuproc() will terminate when clkcountermsec 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 procio(void), in system/procio.c, that implements a while-loop to check if clkcountermsec has exceeded STOPCOND. If so, procio() terminates. Unlike the body of cpuproc()'s while-loop which is empty, procio()'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 3 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 3 msec. Inspect the value of clkcountermsec before and after the for-loop to calibrate the bound.

Benchmark output. Before terminating, cpuproc() and procio() print PID, "CPU-bound" or "I/O-bound" depending on type, CPU usage by calling totcpu(), response time by calling avgresponse(), and clkcountermsec. Modify nulluser() so that in the infinite while-loop it checks if clkcountermsec has exceeded STOPCOND. If so, nulluser() outputs PID, "idle process", CPU usage, response time, and clkcountermsec. Code nulluser() so that it does this only once. As before nulluser() does not terminate.

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() which calls chprio() to set its priority to 3. 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. Output by the four app processes upon termination at around 10 seconds of wall time should indicate approximately equal sharing of CPU time and similar response times. Some CPU time will have been consumed by the idle process which will also have a response time. The total sum of CPU times should be approximately 10 seconds. Discuss your results in lab3.pdf.

Benchmark B. Repeat benchmark scenario A with the difference that the app processes execute procio(). Since the apps are homogenous, their CPU usage and response time should be similar. Discuss your finding in lab3.pdf. Compare the results of benchmarks A and B.

Benchmark C. Let the workload generator main() create 8 app processes, half of them executing cpuproc(), the other half procio(). The four CPU-bound processes should get similar CPU usage and response times since they are homogenous. The same goes for the four I/O-bound processes. Across the two groups, CPU-bound processes should get both higher CPU usage and response times than I/O-bound processes. Discuss your finding in lab3.pdf.

Benchmark D. Code an app, void parasite(void), in system/parasite.c that exploits a weakness of the dynamic priority XINU scheduler which classifies a process that calls sleepms(0) as exhibiting I/O-bound behavior. The logic underlying parasite() is to occupy the CPU for slightly less than QUANTUMIO

4.1 Process Classification

- CPU-bound: Process depleted its time slice
- I/O-bound: Hasn't depleted but voluntarily gives up the CPU by making a blocking call
- 3rd case: Process preempted by a higher/equal priority process that was blocked but is ready
- pri16 prprio is 1 for CPU-bound, 2 for I/O-bound, 0 for XINU's idle/null process
- #define QUANTUMCPU 50
  QUANTUMIO 5
  QUANTUMIDLE 100
  in include/process.h ✓

4.2 Upper half blocking & lower half preemption

- Use sleepms() as a representative blocking system call
  - ↳ Process is considered I/O-bound if it calls sleepms() frequently
  - ↳ Calling sleepms() with negative argument returns SYSERR ✓
  - ↳ Argument 0 prompts kernel to call resched() — called by yield()
- Only consider preemption events triggered by clkhandler()
  - ↳ Check if a process in PR_SLEEP needs to be woken up & inserted into readylist
  - ↳ Time slice depletion where clkhandler() calls resched()
- Modify create() so it ignores arg3 & set the priority of new processes to 1 ✓
- For benchmark testing, use main() to generate workload processes & then terminate
  - ↳ Assign main()'s priority to 3 by calling chprio() ✓
- Global variable preempt used to track current process's remaining time slice must be set appropriately for CPU/IO-bound processes & idle processes in the kernel code

4.3 Testing & performance evaluation

- Use XINUDEBUG to suppress output of debug messages in submitted code.
  Use XINUTEST to enable output of the values during benchmarking
- Code void cpuproc(void) in system/cpuproc.c that implements an empty while loop ✓
  - ↳ While loop checks if clkcountermsec exceeds STOPCOND (set to 1000 msec in include/process.h)
  - ↳ A process executing this terminates after 10 seconds ✓
- Code void procio(void) in system/procio.c
  - ↳ Has a while loop to check if clkcountermsec exceeds STOPCOND. If so, terminate ✓
  - ↳ Body of while loop has for-loop followed by sleepms() with argument 50 ✓
    - ↳ Try different bounds such that it consumes several ms (not more than 3) ✓
- Both functions print PID, "CPU"/"I/O" + "-bound", CPU usage (totcpu()), response time (avgresponse()) & clkcountermsec ✓

- Modify nulluser() so in infinite while-loop, it checks if clkcountermsec exceeds STOPCOND
  - ↳ If so, output PID, "idle process", CPU usage, response time & clkcountermsec ✓
  - ↳ Code it so it only does it once ✓

- Benchmark A:
  - ↳ Process running main() spawns 4 app processes each executing cpuproc() ✓
  - ↳ Call resume() to ready the 4 processes after creating them ✓
  - ↳ Process running main() terminates after creating & resuming the 4 processes
  - ↳ Output upon termination should indicate approx. equal sharing of CPU time & similar response times.
  - ↳ Total sum of CPU time should be approx 10 seconds ✓

- Benchmark B:
  - ↳ Repeat A but with procio()
  - ↳ Should have similar CPU usage & response time

- Benchmark C:
  - ↳ main() create 8 processes, half cpuproc() half procio() ✓
  - ↳ Similar process should have similar CPU usage & response times ✓ ✓
  - ↳ CPU-bound processes should get higher CPU usage & response times

- Benchmark D:
  - ↳ Code void parasite(void) in system/parasite.c
    - ↳ To occupy the CPU for slightly less than QUANTUMIO & call sleepms(0)
    - ↳ Appears to kernel as being I/O-bound when it has CPU-bound behavior
  - ↳ Modify yield() such that current process gets fresh time slice QUANTUMIO & priority is set to 2 ✓
  - ↳ Create 3 processes executing cpuproc() & one executing parasite()
  - ↳ parasite() should have most of CPU time

<u>Bonus</u>

- Code procio()

and call sleepms(0) before the kernel has a chance to potentially preempt the process. If all processes in the readylist are CPU-bound, parasite() will get a fresh time slice of QUANTUMIO and continue to run. Hence parasite() appears to the kernel as being I/O-bound when its actual behavior is CPU-bound. Create three processes that execute cpuproc() and one process that executes parasite(). After all processes terminate at around 10 seconds the process executing parasite() should have received the lion's share of CPU time. Discuss your results in lab3.pdf.

Bonus problem [20 pts]

Modify the I/O-bound app, procio(), into a variant, prociogang(), in system/prociogang.c so that a group of I/O-bound processes can easily starve a CPU-bound process. Create three processes executing prociogang() and one process executing procpu(). When they terminate at around 10 seconds of wall clock time, the three I/O-bound processes should have received approximately equal performance while starving the CPU-bound process. Describe the changes you made in prociogang() and discuss your results.

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 #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-fall2023/compile directory and run "make clean".

ii) Go to the directory where lab3 (containing xinu-fall2023/ and lab3.pdf) is a subdirectory.

For example, if /homes/alice/cs354/lab3/xinu-fall2023 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