CS 354 Spring 2024

Lab 3: Process Coordination and Synchronization [200 pts]

Due: 3/1/2024 (Fri.), 11:59 PM

1. Objectives

The objectives of this lab are to a) understand how semaphores can coordinate producer-consumer with a circular buffer, and b) provide deadlock detection service for semaphore system calls.

---

2. Readings

1. Xinu setup from previous labs
2. Read Chapter 7 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. Producer-consumer synchronization [100 pts]

We will implement scenarios where producer and consumer processes coordinate with semaphores. Also, both producer and consumer processes will concurrently access one circular buffer that is shared among them.

3.1 Circular buffer

First, we will define a circular buffer in include/cb.h with a global, static array of characters (named cbuffer) of size BUF_SIZE (50) and two global head/tail pointers: uint32 cb_head and uint32 cb_tail.

Note: Make sure to include any new header file in include/xinu.h; otherwise, any declaration placed in the header file will not have any effect.

Also, implement all of the following functions in one file system/cb.c:

- void cb_init()

  Place any code that is related to initializing a circular buffer. Call this function once from system/initialize.c.

- void cb_insert(char ch)

  It adds one character ch into the head of cbuffer. cb_head should be incremented accordingly. Note that cbuffer is circular, and thus once cb_head reaches the end of cbuffer, it needs to be reset to 0.

- char cb_remove()

  It removes one character from the tail of cbuffer and returns it. cb_tail should be incremented accordingly.

3.2 Producer and consumer processes

First, define any data structure (e.g., semaphores) necessary for coordinating producer and consumer processes in include/pc.h.

Next, implement the following system calls:

- void pc_init()

  Implement in system/pc_init.c. Place any code that is related to initializing data structures specified in include/pc.h. Call this function once from system/initialize.c.

- void pc_produce(char* str, uint32 len)

Implement in system/pc_produce.c. Once called, the first len number of characters (starting from the one pointed by str) will be inserted to cbuffer. We assume that the total length of string str (including a null terminator) will be equal or greater than len. Use cb_insert(ch) to insert a character to cbuffer one-by-one. Note that the producer process running this function needs to coordinate with other producers/consumers and coordinate over the access of cbuffer.

- void pc_consume(char* buf, uint32 len)

Implement in system/pc_consume.c. It removes len number of characters from cbuffer and writes to buf. We assume that the length of buf is equal or greater than len. Use cb_remove() to remove a character from cbuffer. As similar to pc_produce(), the consumer process running this function needs to coordinate with other producers/consumers and coordinate over the access of cbuffer.

3.3 Test cases

Start with a basic case where main() spawns some producer and consumer processes that call pc_produce() and pc_consume(), respectively. Try not to make cbuffer full. Once you verify that process coordination works as intended, try more complicated cases, such as adding more processes or forcing cbuffer to be full at a certain point of the program.

---

4. Deadlock detection extension of semaphores [100 pts]

In this problem, we will extend the semaphore services exported by XINU such that deadlocks can be detected.

4.1 Semaphore resource allocation graph for deadlock detection

To implement deadlock detection, we will utilize a data structure called a semaphore resource allocation graph (resource graph in short), which consists of nodes and edges as follows:

- Nodes

  A node can either be a process or a semaphore.

- Edges

An edge encodes the information on either a) a semaphore being owned/held by a process, or b) a process requesting/waiting for a semaphore.

Define your own data structures for implementing the resource graph. For any custom data structures (other than extending XINU default data structures), specify them under include/dl.h. There are a few assumptions/restrictions on the resource graph for this lab:

- We will restrict ourselves with static memory allocation. Do not use dynamic memory allocation primitives, such as getmem() in XINU, to implement the resource graph.
- We will only consider the cases where any process can request/wait for at most one semaphore, and any semaphore can be owned/held by at most one process at any time.

4.2 Maintaining resource graph with semaphore system calls

In order to maintain the resource graph, we will implement two new system calls:

- syscall dl_wait(sid32 sem)

Implement in system/dl_wait.c. It is identical to wait() except for two additional constraints over the case when a process is put in a semaphore queue: a) change its state to PR_DLWAIT instead of PR_WAIT. Define the new process state PR_DLWAIT in include/process.h. b) update the resource graph properly.

- syscall dl_signal(sid32 sem)

Implement in system/dl_signal.c. It is identical to signal() except that the resource graph must be updated when a process dequeues from a semaphore and becomes ready.

Note: We will not enforce compatibility with XINU default system calls, wait() and signal(). That is, calls to wait() will only be paired with calls to signal(), and so do the calls to dl_wait() and dl_signal().

4.3 Outputting a deadlock

We will make XINU to check whether any deadlock exists, once every second, and if any exists, output a cycle in the resource graph.

First, implement the following function that will be used for printing a deadlock.

- void print_deadlock()

Implement in system/print_deadlock.c. It iterates over the existing resource graph (how you iterate depends on your data structure). If no deadlock (i.e., cycle in the resource graph) is detected, print: “At %d seconds, no deadlock is detected.\n” where %d is substituted by the value of (uint32) clktime. Otherwise, print: “At %d seconds, a deadlock is detected!\n”. In addition, print any one cycle in the resource graph in an easily readable manner.

Then, modify system/clkhandler.c to call print_deadlock() in a way that XINU snapshots and prints the deadlock status once every second since bootload time, e.g., at clktime = 0, 1, 2, ….

To reduce overhead over the search, we will set both NPROC and NSEM to 20 for this lab. Two values can be adjusted as we have done in lab 1 (modify Configuration file under config/).

4.4 Test cases

First, start with a basic scenario having 2 processes and 2 semaphores. Carefully craft a scenario in a way that a deadlock occurs, that is, make it so that specific calls to dl_wait() or dl_signal() occur in a specific sequence. Second, run 3 processes and 3 semaphores and try a case with a deadlock.

Bonus problem [25 pts]

Problem 4 introduced a deadlock detection mechanism, but it is still under question what to do when a deadlock occurs.

Consider a hypothetical scenario where dl_wait() is modified in a way that it returns an error (say, SYSERR_DL) when calling dl_wait() would result in a cycle in the resource graph (instead of blindly allowing the cycle). Assume that a developer has built some program, and it turns out a call to dl_wait() from some process has returned SYSERR_DL. What would be a rational way to fix this problem from the developer's perspective? Provide one approach and explain your reasoning in lab3.pdf (place the pdf under lab3/ directory). Try to be brief and concise (e.g., aim for one or two paragraphs). Basic approaches, such as terminating the offending process, or sleeping and then attempting to call dl_wait(), will not be accepted.

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 lab3 (containing xinu-spring2024/) is a subdirectory.

For example, if /homes/alice/cs354/lab3/xinu-spring2024 is your directory structure, go to /homes/alice/cs354

iv) 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.