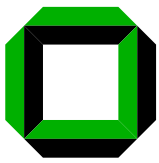

University of Karlsruhe
System Architecture Group
Gerd Liefländer

Teaching Assistant:
Philipp Kupferschmied

System Architecture 2008/09
Programming Assignment 1 Update 1

Submission Deadlines: 21.11.2008, 12:00h (theoretical part)
12.12.2008, 12:00h (practical part)

This document contains both the theoretical and the practical assignments. You might want to read the entire document once before reading it again to answer the questions. The next step is then to analyze the problems and write a simple design document. Afterwards, as the last step, start coding and submit your solution—on time.

1 Introduction

In this assignment you will solve a number of synchronization and locking problems. You will also get experience with data structure and resource management issues.

2 Write Readable Code

In your programming assignments, you are expected to write well-documented, readable code. There is a variety of reasons to strive for clear and readable code: Code that is understandable to others is a requirement for any real-world programmer, not to mention the fact that, after enough time, you will be in the shoes of one of the others when attempting to understand what you wrote in the past. Finally, clear, concise, well-commented code makes it easier for the assignment marker to award you marks! (This is especially important if you cannot get the assignment running. If you cannot figure out what is going on, how do you expect us to do it?)
There is no single right way to organize and document your code. It is not our intent to dictate a particular coding style for this class. The best way to learn about writing readable code is to read other people's code, for example OS/161. When you read someone else's code, note what you like and what you do not like. Pay close attention to the lines of comments which most clearly and efficiently explain what is going on. When you write code yourself, keep these observations in mind.
Here are some general tips for writing better code:

- Split large functions. If a function spans multiple pages, it is probably too long.
- Group related items together, whether they are variable declarations, lines of code, or functions.
- Use descriptive names for variables and procedures. Be consistent with this throughout the program.
- Comments should describe the programmer's intent, not the actual mechanics of the code. A comment which says "Find a free disk block" is much more informative than one that says "Find first non-zero element of array."

3 Groups

We assume that the assignment has been completed by all group members equally, i.e., all of you have done the same amount of work. Please hand in the assignment before the deadline for the submission has passed or your submission will not be marked.

4 Setting up the Assignment

We will now guide you through setup required for this assignment.

4.1 Obtaining and setting up ASST1 in Mercurial

In this section, we will create a new, shared repository that contains the source code of assignment 1. In order to do so, *s_hghost* has to perform the following commands

```
$ mkdir -p ~/sysarch/sharedrepos/asst1-src
$ cd ~/sysarch/sharedrepos/asst1-src
$ hg init
$ cd ~/sysarch/
$ hg clone sharedrepos/asst1-src
$ wget http://i30www.ira.uka.de/~pkupfer/edu/2008ws/sysarch/asst1-src.tbz2
$ tar -xjf asst1-src.tbz2
$ rm asst1-src.tbz2
$ cd ~/sysarch/asst1-src
$ hg add
$ hg commit
$ hg tag asst-base
$ hg push
```

4.1.1 Obtaining a Working Copy

s_hghost has already cloned the shared repository during the setup process described above. *s_member* can clone it via

```
$ mkdir -p ~/sysarch
$ cd ~/sysarch
$ hg clone ssh://sysarch_hg/asst1-src
```

After having performed these steps, both *s_hghost* and *s_member* have a local copy of the repository in `~/sysarch/asst1-src`.

5 Begin your Assignment

You can start by making an initial build of the provided code.

5.1 Configuring ASST1

Before proceeding any further, configure the sources via

```
$ cd ~/sysarch/asst1-src
$ ./configure --ostree="$HOME/sysarch/root"
```

We have provided you with a framework to run your solutions for ASST1. This framework consists of driver code (found in `kern/asst1`) and menu items you can use to call your solutions from the OS/161 kernel menu.
You also have to configure your kernel before you can use this framework. The procedure for configuring the kernel is the same as in ASST0:

```
$ cd ~/sysarch/asst1-src/kern/conf
$ ./config ASST1
```

You should see an `ASST1/` directory in `~/sysarch/kern/compile`.

5.2 Building ASST1

When you built OS/161 for ASST0, you ran make from `kern/compile/ASST0`; in ASST1, you use:

```
$ cd ~/sysarch/asst1-src/kern/compile/ASST1
$ make
```

If you are told that the `compile/ASST1` directory does not exist, make sure you ran config for ASST1. Run the resulting kernel.

```
$ cd ~/sysarch/root
$ ./sys161 kernel
sys161: System/161 release 1.13, compiled Oct 12 2007 16:36:52
OS/161 base system version 1.10
Copyright (c) 2000, 2001, 2002, 2003
President and Fellows of Harvard College. All rights reserved.
Put-your-group-name-here's system version 0 (ASST1 #1)
Cpu is MIPS r2000/r3000
2048k physical memory available
Device probe...
lamebus0 (system main bus)
[...]
pseudorand0 (virtual)
OS/161 kernel [? for menu]:
```

In order to execute the tests in this assignment, you need more than 512 kB of RAM typically used by System/161. We suggest that you allocate at least 2 MB (but no more than 16 MB) to System/161 by editing `~/sysarch/root/sys161.conf`. Change the line for the `busctl` device to read

```
31 busctl ramsize=2097152  # 2 MB of RAM
```

5.3 Command Line Arguments to OS/161

Your solutions to ASST1 will be tested by running OS/161 with command line arguments that correspond to the menu options in the OS/161 boot menu.
IMPORTANT: DO NOT change these menu option strings!
Here are some examples of using command line arguments to select OS/161 menu items:
`$ ./sys161 kernel ''at;bt;q''`.
This is the same as starting up with `./sys161 kernel`, then typing `at` at the menu prompt (invoking the array test), then typing `bt` (bitmap test), and finally quitting by typing `q`.
`$ ./sys161 kernel ''q''`
This is the simplest example, which will start the kernel and quit as soon as it has finished booting. Try it yourself with other menu commands. Remember that the commands must be separated by semicolons (`;`).

6 Concurrent Programming with OS/161

If your code is properly synchronized, the timing of context switches and the order in which threads run should not change the behavior of your solution. Of course, your threads may print messages in different orders, but you should be able to easily verify that they follow all of the constraints applied to them and that they do not deadlock.

6.1 Built-In Thread Tests

When you booted OS/161 in ASST0, you may have seen the options to run the thread tests. The thread test code uses the semaphore synchronization primitive. You should trace the execution of one

of these thread tests in GDB to see how the scheduler acts, how threads are created, and what exactly happens in a context switch. You should be able to step through a call to `mi_switch()` and see exactly where the current thread changes.
Thread test 1 (`tt1` at the prompt or on the kernel command line) prints the numbers 0 through 7 each time each thread loops. Thread test 2 (`tt2`) outputs only when a thread starts or exits. The latter is intended to show that the scheduler does not cause starvation—the threads should all start together, spin for a while, and then end together.

6.2 Debugging Concurrent Programs

`thread_yield()` is automatically called for you at intervals that vary randomly. While this randomness is fairly close to reality, it complicates the process of debugging your concurrent programs. The random number generator used to vary the time between calls of `thread_yield()` uses the same seed as the random device in System/161. This means that you can reproduce a specific execution sequence by using a fixed seed for the random number generator. You can pass an explicit seed into the random device by editing the "random" line in your `~/sysarch/root/sys161.conf` file. For example, to set the seed to 1, you would edit the line to look like: `28 random seed=1`.
We recommend that while you are writing and debugging your solutions you pick a seed and use it consistently. Once you are confident that your threads do what they are supposed to do, set the random device to `autoseed`. This should allow you to test your solutions under varying conditions and may expose scenarios that you had not anticipated. To reproduce your test cases, you additionally need to run your tests via command line arguments to `sys161` as described above.

7 Question Exercises

Please answer the following questions and submit the answers to (a subset of) them before the deadline online on the submission form of the website. Some of the questions will not be assessed and are only provided to guide you and support your understanding of the supplied code.

7.1 Code Reading

To implement synchronization primitives, you will have to understand the operation of the threading system in OS/161. When you are writing solution code for the synchronization problems, it will help if you also understand exactly what the OS/161 scheduler does when it dispatches among threads.

7.1.1 Threads

Question 1.1: What happens to a thread when it exits (i.e., calls `thread_exit()`?
Question 1.2: What happens to a thread when it sleeps?
Question 1.3: Which function(s) handle(s) a context switch?
Question 1.4: How many thread states are defined in OS/161?
Question 1.5: What does it mean to turn interrupts off?
Question 1.6: How is this accomplished?
Question 1.7: Why is it important to turn off interrupts in the thread subsystem code?
Question 1.8: What happens when a thread wakes up another thread?
Question 1.9: How does a sleeping thread get to run again?

7.1.2 Scheduler

Question 1.10: Which function is responsible for choosing the next thread to run?
Question 1.11: How does that function pick the next thread?
Question 1.12: What role does the hardware timer play in scheduling?
Question 1.13: What hardware independent function is called on a timer interrupt?

7.1.3 Synchronization

Question 1.14: Describe how `thread_sleep()` and `thread_wakeup()` are used to implement semaphores.
Question 1.15: What is the purpose of the argument passed to `thread_sleep()`?
Question 1.16: Why does the lock API in OS/161 provide `lock_do_i_hold()` but not `lock_get_holder()`?

7.2 Synchronization Problems

The following problems are designed to familiarize you with some of the problems that arise in concurrent programming and help you learn to identify and solve them.

7.2.1 Identify Deadlocks

Consider the following code fragments:

```
semaphore *mutex, *data;

void me(void) {
    P(mutex);
    /* do something */
    P(data);
    /* do something else */
    V(mutex);
    /* clean up */
    V(data);
}

void you(void) {
    P(data)
    P(mutex);
    /* do something */
    V(data);
    V(mutex);
}
```

Question 1.17: Propose a change to one or both of them that makes deadlocks impossible. Which general principle does the original code violate, causing a potential deadlock?

7.2.2 More Deadlock Identification

Now consider this code fragment:

```
lock *file1, *file2, *mutex;

void laurel(void) {
    lock_acquire(mutex);
    /* do something */
    lock_acquire(file1);
    /* write to file 1 */
    lock_acquire(file2);
    /* write to file 2 */
    lock_release(file1);
    lock_release(mutex);
    /* do something */
    lock_acquire(file1);
    /* read from file 1 */
```

```
    /* write to file 2 */
    lock_release(file2);
    lock_release(file1);
}

void hardy(void) {
    /* do stuff */
    lock_acquire(file1);
    /* read from file 1 */
    lock_acquire(file2);
    /* write to file 2 */
    lock_release(file1);
    lock_release(file2);
    lock_acquire(mutex);
    /* do something */
    lock_acquire(file1);
    /* write to file 1 */
    lock_release(file1);
    lock_release(mutex);
}
```

Question 1.18: Can two or more threads executing `laurel()` and `hardy()` deadlock?

7.3 Synchronized Queues

Question 1.19: The thread subsystem in OS/161 uses a queue structure to manage some of its state. This queue structure is not synchronized. Why not?

Question 1.20: Under what circumstances should you use a synchronized queue structure?

You can submit your answers on the website, we will select 15 of the above questions to answer online.

8 Coding Assignment

We know that you have been itching to get to the coding. Well, you have finally arrived!
The following problems will give you the opportunity to write two fairly straightforward concurrent programs and get a more detailed understanding of how to use concurrency mechanisms to solve problems.
We have provided you with basic driver code that starts a predefined number of threads that execute a predefined activity (in the form of calling functions that you must implement). You are responsible for implementing the called functions. Remember to specify a seed to use in the random number generator by editing your `sys161.conf` file, and run your tests using System/161 command line arguments. It is much easier to debug initial problems when the sequence of execution and context switches is reproducible. When you configure your kernel for ASST1, the driver code and extra menu options for executing your solutions are automatically compiled in.

8.1 Concurrent Mathematics Problem

For the first problem, we ask you to solve a very simple mutual exclusion problem. The code in `kern/asst1/math.c` counts from 0 to 10000 by starting several threads that increment a common counter. You will notice that—as supplied—the code operates incorrectly and outputs garbage like `345 + 1 = 352`.
Once the count of 10000 is reached, each thread signals the main thread that it is finished and exits. Once all `adder()` threads exit, the main thread `math()` cleans up and exits, too.
Your job is to modify `math.c` by placing synchronization primitives appropriately so that incrementing the counter works correctly. The statistics printed should also be consistent with the overall count.

Note that the number of increments each thread performs is dependent on scheduling and hence will vary. However, the total should equal the final count. To test your solution, use the `1a` menu choice. Sample output from a correct solution follows:

```
$ ./sys161 kernel "1a;q"
sys161: System/161 release 1.13, compiled Oct 12 2007 16:36:52
OS/161 base system version 1.10
Copyright (c) 2000, 2001, 2002, 2003
President and Fellows of Harvard College. All rights reserved.
Put-your-group-name-here's system version 0 (ASST1 #28)
Cpu is MIPS r2000/r3000
2048k physical memory available
Device probe...
lamebus0 (system main bus)
[...]
pseudorand0 (virtual)
OS/161 kernel: 1a
Starting 10 adder threads
Adder threads performed 10000 adds
Adder 0 performed 1008 increments.
Adder 1 performed 1032 increments.
Adder 2 performed 998 increments.
Adder 3 performed 1017 increments.
Adder 4 performed 1012 increments.
Adder 5 performed 988 increments.
Adder 6 performed 971 increments.
Adder 7 performed 975 increments.
Adder 8 performed 1027 increments.
Adder 9 performed 972 increments.
The adders performed 10000 increments overall
Operation took 3.665222400 seconds
OS/161 kernel: q
Shutting down.
The system is halted.
```

8.2 Restaurant Synchronization Problem

Suppose you were working your way through university and decided to take a job in a restaurant seating customers. The restaurant is next to the famous "tourist trap" monument, and the main clientele are bus loads of tourists, managed by various tour operators. You start your first day and find complete chaos: Customers are not being seated or are being seated at the wrong tables. Some customers are fighting over tables, requests for tables are getting lost. Some customers are waiting forever for their table, while others seem to get all the service.
Being an operating system expert, you quickly realize that the restaurant's problems are related to concurrency issues between the requesting of tables by tour operators and the allocation of tables. You volunteer your services to provide a solution to the restaurant's problems, reduce the chaos, and restore order.

8.2.1 The Basic Restaurant

To provide a solution, you must come to terms with the basic elements of the restaurant that you have to work with. The restaurant consists of a set of tables labeled 1 to N (where N is the maximum table number of the day). Some tables have better views than others, so the restaurant allows the tour operators to request the set of tables they would like for each of their bus loads of tourists. All members of each bus load request their tables, are seated when the tables become available, eat, and leave. The

basic elements are defined in `kern/asst1/restaurant_driver.h`. The actions of the tour operators are defined in `kern/asst1/restaurant_driver.c`. Have a look at these files for detailed comments. For each tour bus a tour operator manages, he

- builds a list of requested tables from the current bus load of tourists he manages
- submits the request to the restaurant and waits for tables to become available
- seats his tourists, and they eat
- indicates to the restaurant when the tourists have finished eating and the table(s) are free again

The function `run_restaurant()` is called via the menu in OS/161 (item `1b`); its purpose is as follows:

- It initializes all the table counters to zero uses.
- It calls `restaurant_open()`, a routine you will provide to set up the restaurant.
- It then creates some threads to run as tour operators. Note that these threads obviously run concurrently.
- The driver thread then waits on a semaphore for all the tour operators to finish, after which we print out the table statistics for the day.
- Finally, it calls `restaurant_close()`, a procedure you provide to clean up when the restaurant has closed.

Have a quick look through both `restaurant_driver.c` and `restaurant_driver.h` to reinforce your understanding of what is going on (well, at least what is *expected* to go on).
Your job is to write the functions outlined in `restaurant.c` that perform most of the work. Each function is described in `restaurant.c`. Generally, your solution must result in the following when `run_restaurant()` is called during testing:

- The restaurant is prepared for opening.
- All tour operators have their tables allocated correctly.

  "Correct" means that each non-null entry in the `table_request` array results in tourists of the indicated operator having exclusive access to the requested tables—until they leave.
- The restaurant is suitably cleaned up afterwards (allocated memory or locks, semaphores, ... are freed).
- Statistics kept on table usage are consistent with the requests made.

You can modify `restaurant_driver.c` and `restaurant_driver.h` to test different scenarios (e.g., vary the number of tour operators or tables requested), but your solution must not rely on any changes you make to the `restaurant_driver.c` file.
You will have to modify `restaurant.c` to implement your solution. However, your modifications have the constraint that they must still work with an original `restaurant_driver.c`. For testing, we will replace `restaurant_driver.c` and `.h` with logically equivalent versions that may vary the numbers of participants, and the tables requested. We may also vary the timing of various functions. A correct solution will work for all variations we test. Sample output from a correct solution is included below:

```
$ ./sys161 kernel "1b;q"
sys161: System/161 release 1.13, compiled Oct 12 2007 16:36:52
OS/161 base system version 1.10
Copyright (c) 2000, 2001, 2002, 2003
President and Fellows of Harvard College. All rights reserved.
Put-your-group-name-here's system version 0 (ASST1 #32)
Cpu is MIPS r2000/r3000
```

```
2048k physical memory available
Device probe...
lamebus0 (system main bus)
[...]
pseudorand0 (virtual)
OS/161 kernel: 1b
Opening the restaurant
The tour operators are starting
OP 0 starting
OP 1 starting
OP 2 starting
OP 3 starting
Op 0 going home after seating 200 tables
OP 4 starting
Op 1 going home after seating 200 tables
Op 2 going home after seating 200 tables
Op 3 going home after seating 200 tables
Op 4 going home after seating 200 tables
The restaurant is closing
Table 1 reserved 1000 times
Table 2 reserved 0 times
Table 3 reserved 0 times
Table 4 reserved 0 times
Table 5 reserved 0 times
Table 6 reserved 0 times
Table 7 reserved 0 times
Table 8 reserved 0 times
Table 9 reserved 0 times
Table 10 reserved 0 times
Table 11 reserved 0 times
Table 12 reserved 0 times
Table 13 reserved 0 times
Table 14 reserved 0 times
Table 15 reserved 0 times
Table 16 reserved 0 times
Table 17 reserved 0 times
Table 18 reserved 0 times
Table 19 reserved 0 times
Table 20 reserved 0 times
Total reservations 1000
The restaurant is closed, bye!!!
Operation took 0.555521080 seconds
OS/161 kernel: q
Shutting down.
The system is halted.
```

Warning: The driver as provided by us only only allocates one table per bus load, it requests always table #1, and does not demonstrate any corner cases at all. Make sure to test your solution with more `random()` drivers with (at least) varying (a) numbers of tours per operator, (b) numbers of operators, (c) numbers of requested tables per tour, and (d) numbers of tables in use (using the first 1, 2, 3, 5, 10, or 20 tables) before submitting—we will!

8.2.2 Before Coding

You should have a very good idea of what you are attempting to do before you start. Concurrency problems are very difficult to debug, so it is in your best interest that you convince yourself you have a correct solution before you start implementing it.
The following questions may help you develop your solution:

- What are the shared resources?
- Who shares which resources?
- Who produces what? And who consumes it?
- What states can the various resources be in?
- What do you need to keep a count of in the restaurant?
- How does your solution prevent deadlock or starvation?

Try to frame the problem in terms of resources requiring concurrency control, and producer/consumer problems. A diagram may help you to understand the problem.

9 Evaluating your Solution

Your solution will be judged in terms of its correctness, conciseness, clarity, and performance. Performance will be judged in at least the following areas.

- Are all computations correctly synchronized?
- Do all the tour operators participate?
- Can tourists from different operators eat in parallel if they do not require the same tables?
- Do you define critical sections larger than needed?

10 Documenting your Solution

This is a compulsory aspect of the assignment! You must write a small design document identifying the basic issues in both of the concurrency problems in this assignment, and then describe your solution to the problems you have identified. For example, detail which data structures are shared, and which code forms a critical section.
The document must be plain ASCII text. We expect such a document to be roughly 200–600 words, i.e., clear and to the point. The document will be used to guide our markers in their evaluation of your solution to the assignment. In the case of poor results in the functional testing combined with a poor design document, we will base our assessment on these components alone. If you cannot describe your own solution clearly, you cannot expect us to reverse engineer the code to a poor and complex solution to the assignment.
Create your design document at the top of the source tree to OS/161 (i.e., in `~/sysarch/asst1-src`), and include it in the Mercurial repository as follows.

```
$ cd ~/sysarch/asst1-src
$ hg add design.txt
```

When you later commit your changes into your repository, your design document will be included in the commit, and later in your submission. Also, please word wrap your design doc if your have not already done so. You can use GNU `fmt` to achieve this if your editor does not.

11 Assignment Submission

As with the previous assignment, you will again submit a diff of your changes to the original sources. Both team members therefor commit their latest changes locally and push them back to the shared repository:

```
$ cd ~/sysarch/asst1-src
$ hg commit
$ hg push
```

Manually resolve conflicts if necessary, and ensure that your final version is still correctly working. Again, one of you has to create the diff output and send it to your tutor. To do so, he/she ensures that his/her local copy of the repository is up-to-date

```
$ cd ~/sysarch/asst1-src
$ hg pull
$ hg update
```

If everything is all right, the latest revision can be tagged, and the diff can be created:

```
$ cd ~/sysarch/asst1-src
$ hg tag asst-final
$ hg diff -r asst-base -r asst-final > ~/asst1.diff
```

Make sure that you have commited your changes before tagging the final revision and before creating the diff! Have a look at asst1.diff before you submit it, e.g. with `less ~/asst1.diff`. If you can't find the code you wrote in the diff-file, it is very likely that something went wrong.

11.1 Submitting your Assignment

Send the diff (`~/asst1.diff`) of your solutions via email to `os161@ira.uka.de`. Include your group name and assignment number in the subject line.
You're now done.
Note: If for some reason you need to change and re-submit your assignment after you have tagged it "asst-final", you will need to either delete the "asst-final" tag, commit the new changes, re-tag, and re-diff your assignment, or choose a different final tag name and commit the new changes, tag with the new tag, and re-diff with the new tag. To delete a tag, use `hg tag --remove` *tagname*.

12 If you want more...

If the assignment was too easy or you are done too early, you are free to do some additional work. This will not be graded and should not be submitted along with the above diff! But this will definitely give you a better understanding of OS internals. Please do this only after you have submitted the assignment.
One suggestion would be to implement semaphores or locks/mutexes yourself. Simply delete the inner workings of `P()`, `V()`, `acquire_lock()`, and/or `release_lock()` without looking at the implementation and try to implement the functions yourself. You may need to disable and reenable interrupts and take appropriate actions. This can be quite tricky, thus this optional part is ... optional and not graded! If you look around the code you may as well find other interesting areas to play with.
Once you are done reimplementing semaphores, you may write simple tests, or test them with the existing code (e.g., `math.c`). Then you can compare your code to the original implementation. Again, this is not required and not graded. Please do **not** submit the optional parts; they are just hints: Implementing semaphores may well be more exciting than simply using them.