1 Overview

DLXOS has been changed to fit the requirements for this project. You will have to copy this new version of DLXOS (`/home/cs314/project2.tgz`) to see the changes.

1.1 Setup Instructions

As before:

```
mkdir ~/cs314/project2
cp /home/cs314/project2.tgz ~/cs314/project2/
cd ~/cs314/project2/
tar xvfz project2.tgz
```

Make sure you have `/home/cs314/dlxtools/bin/` in your PATH.
Compile as before : type `make` in `~/csc314/project2/src` directory.
Run in `~/csc314/project2/execs` directory

1.2 DLXOS Modifications

DLXOS has been modified as follows:

User Programs: The `main()` routine in process.c has been modified to load the user-specified executable program with user-specified command-line parameters. Suppose you have a user program called `userprog.dlx.obj` (it can be any program, the name is not important), you can run it using the following command:

```
dlxsim -x os.dlx.obj -a -u userprog.dlx.obj [param1 param2 ...]
```

where `param1, param2, ...` are the optional command-line parameters to the user program. Note that the name of the user program must follow the -u option, and the rest of the parameters from the command line are passed to the user program as command-line arguments. If you have never written any C programs that take command-line arguments, now is the time to find that dusty C book.

New APIs: DLXOS does not support threads. In order to solve some of the problems in this project, several processes need to share some memory, e.g. used for the shared buffer in the producer-consumer solution. Provided here is a primitive implementation of shared memory support for your

use. In addition, there is also user-level access to semaphores, and support for starting a user process from a user program. See the documentation in the API section.

Examples on using the new APIs: There are two example C files: userprog.c and userprog2.c. The Makefile is modified in such a way that these files will automatically be compiled when you type make in the directory, and will then be moved to the directory `../execs`. To execute the example, you need to type (in the execs directory)

```
dlxsim -x os.dlx.obj -a -u userprog.dlx.obj number
```

where number is any number between 1 through 10. The program in `userprog.c` creates a shared memory page and two semaphores. It then creates number instances of program in `userprog2.c` and passes the shared page handle, the two semaphores, and number as command-line arguments to each instance. The program in `userprog2.c` in turn retrieves these parameters, maps the memory page to its memory space, prints the number that you provided, and exits. These files will help you understand how to use the provided APIs. Note that there are no threads in DLXOS!!! What you have instead is this ability to share memory.

2 Assignment

- Read queue.c, queue.h, synch.h, synch.c and answer the following questions. Write your answers in design.txt.
  1. (6 points) How many semaphores are available in DLXOS?
  2. (6 points) How many locks are available in DLXOS?
  3. (6 points) How is a semaphore created in DLXOS?
  4. (6 points) Explain how SemSignal and SemWait work?
  5. (6 points) What is a handle and how is it used to access semaphores in DLXOS?
- (35 points) Using only semaphores already implemented in the system implement locks (locks behave as mutexes do, except they don't have to be initialized to a specific value, and multiple unlocks have no effect save for perhaps the first one). Looking at the definition of sem_t, sem_create(int count), SemHandleWait(sem_t sem), SemHandleSignal(sem_t sem), from the API that is provided will give you some clues about this part. Note that it is not a good idea to accept or provide pointer to an OS-level construct from or to a user process, as it can cause security loopholes. Hence we always deal with handles, and not pointers. All your implementations should be done within synch.c and synch.h by filling up the missing parts. The functions should have all the functionality explained in those files. After your implementation of this part is complete, any user program that calls these functions should be able to execute correctly. Document your implementation and testing in design.txt.

- (35 points) Martian scientists have asked us to help out with writing an atmosphere monitoring program. They require a program that:
  - Injects atoms of Nitrogen and Oxygen into the atmosphere
  - Monitors the formation of the O2 and N02 molecules
  - Prints out messages at every stage

  There are 2 primary reactions that the martians are concerned with:

  1. `O + O -> O2`
  2. `N + O2 -> NO2`

  The first reaction should happen when there are at least two Oxygen **atoms** available (and "`O + O -> O2`" should be printed), while the second reaction should happen when there are at least one Nitrogen **atom** and one Oxygen **molecule** available (and "`N + O2 -> NO2`" should be printed).

  In order to solve this problem, you may think of each atom as a separate thread. Every time a Nitrogen atom is created, you will have to print out the message "An atom of Nitrogen was created" Similarly, every time an atom of Oxygen is created, print out the message "An atom of Oxygen was created".

  Here is one method of implementing the solution:

  1. Each N atom invokes a procedure called nReady when it's ready to "react" and prints out "An atom of Nitrogen was created".
  2. Similarly, each O atom invokes oReady when it's ready to react. Print out "An atom of Oxygen was created"
  3. A procedure called makeOxygen should be called when a molecule of Oxygen is formed.

  Note: You may think of makeOxygen as being both producers and consumers (since they "consume" 2 atoms to make a molecule of O2 and they then "produce" one molecule of O2 for the production of NO2). Another procedure called makeNO2 "consumes" 1 atom of Nitrogen and and 1 molecule of Oxygen O2 and "produces" 1 molecule of Nitogen diOxide. The trick is to synchronize appropriately between all of these procedures.

  Use semaphores and/or locks for synchronization. It would make sense to prototype your code before attempting a solution in DLXOS's world. So, take a look at https://computing.llnl.gov/tutorials/pthreads/ and first devise a solution using pthreads where mutexes are managed via `pthread_mutex_lock` and `pthread_mutex_unlock`. For semaphores (mutexes are going to be useless for atomic counters), use POSIX semaphores (on the os server, see `man sem_init`, `man sem_wait`, `man sem_post`, and `man sem_destroy`). In both DLX and pthreads, you will need a process that is going to spawn some number of specified producers and consumers. Remember that you need at least one of each to make progress, and to compile with the -pthreads flag. Turning in a working prototype developed using pthreads in C for this part of the project could be worth up to 15% of the project grade. Not turning in a working prototype does not hurt your grade, if your solution works in DLX.

3 What to Turn In

Turn in 3 things as a group, and 1 item individually:

- group.txt: containing a list of the group members.
- design.txt: containing your answers to the assignment questions and solution design descriptions.
- A .tgz of your project directory, as before. (Include a design.txt file in which you explain where you made your changes and what the changes are intended to do. Any additional information that would be useful for grading: testing, configuration, compiling, etc should go into a README file.)
- review.txt: INDIVIDUALLY, you should turn in a text file containing a brief account of your group activity. Keeping in mind your group's dynamic, answer at least the following questions: What worked? What did not work? What were you responsible for in the project? What could be done differently next time?

4 The API

4.1 User Programs

Prototype: `extern int process_create(char *arg1, ...)`

Usage: `process_create(program name, param1, param2, ...paramn, NULL);`

Description: The function process_create() creates a user process which runs the program_name and passes the rest of the arguments as command-line parameters to the user program. The parameters param1, param2, . . . are passed in the form of character strings. These parameters can be retrieved from the main(int argc, char *argv[]) function of the user program according to normal C convention. Thus, in the user programs, argc denotes the number of command-line arguments (including the program_name) passed to process_create(). Also argv[0] is a character string which contains the program name, argv[1] is a character string which contains the first command-line argument, and so on.

Limitations: The number of parameters to process_create must be less than 128. Also the total length of all the parameters, including the terminating `\0` characters must be less than 1024. Also, the argument list has to be terminated by a NULL pointer (see Usage). The results are unpredictable if this NULL is omitted.

4.2 Shared Memory

Prototype: `uint32 shmget()`

Usage: `handle = shmget();`

Description: shmget() allocates a shared page (page size 64K) in the physical memory and returns a system-wide unique handle that correspond s to this page. The page is also automatically mapped to the virtual address space o f the calling process. The handle as it is cannot be used to access the shared page, but needs to be passed to shmat() (see below) to get a virtual address corresponding to this page.

Limitations: At most 32 shared pages can exist in the system at any given time. Also, any process cannot have more than 16 pages allocated to it (including the shared and the non-shared pages). Under such circumstances (i.e. if the total number of shared pages is equal to 32, or the calling process is already using 16 pages), the function returns the value 0, indicating that a shared page was not allocated. The DLXOS shared-memory API does not support any protection. All pages created using shmget() have read-write permissions.

Prototype: `void *shmat(uint32 handle)`

Usage: `ptr = shmat(handle);`

Description: shmat() maps the unique shared-memory page (page size 64K) identified by the handle to the address space of the calling process, and returns the address that points to the base of this page. If the memory page identified by handle is not a shared page, shmat() does not do any mapping, and returns the NULL pointer. Also, if the calling process is already using 16 memory pages, shmat() does not do any mapping, and returns the NULL pointer. If the page pointed to by handle is already mapped to the address space of the calling process, shmat()returns the address of the base of this page without any further re-mapping. A page mapped to the address space of a process remains mapped until the process exits by making a call to exit(). A page that is not mapped to the address space of any process is freed and added to the list of the free pages. Hence the process that creates a shared page should not quit till some other process maps the shared page to its address space.

Limitations: DLXOS shared-memory API does not support protection . Any shared page in the system can be mapped by any process to its address space. Also, once a page is mapped, the calling process has both read and write permissions to this page. Permissions cannot be given selectively.

4.3 Semaphores

Prototype: `sem_t SemCreate(int count)`

Usage: `sem = SemCreate(int count);`

Prototype: `int SemHandleWait(sem_t sem)`

Usage: `retval = SemHandleWait(sem);`

Prototype: `int SemHandleSignal(sem_t sem)`

Usage: `retval = SemHandleSignal(sem);`

5 FAQ

Q: Where should we put design.txt?

A: Please put a design.txt in the directory of each question. In design.txt, briefly explain how you solve the problem.

Q: Is anyone having a problem when you take out Printf statements in your code it stops working or the process exits to early? It seems like our code works fine when we are debugging but then when we take the printf statements out everything goes south.

A: If you have code like:

```
while (cond == false);
```

Change it to

```
while (cond == false)
{
   Printf("");
}
```

Q: My code works when I have "Printf" in the loop but fails to work when i remove the "Printf".

A: The compiler optimizes (keeps in register) some of the frequently accessed variables. This causes problems when you are trying to synchronize processes using a shared variable. You can resolve this by making all shared variables *volatile*. *Volatile* variables are not optimized by compiler.

Q: What modifications can I make to the Makefile to make my life easier?

A: Refer to GNU's make manual (http://www.gnu.org/software/make/manual/) or find one of the many tutorials online (like http://mrbook.org/tutorials/make/).

Q: Is there a way to use standard libraries like `string.h`? When I use `include <string.h>`, the compiler doesn't seem to know where to look for the string.h file. Any hint?

A: The compiler is customized for dlxos. So it lacks a lot of functions in common libraries. Also Printf() has the following limitations (from traps.c): "This printf code will correctly handle integers but will not correctly handle strings (yet). Also, note that the maximum format string length is 79 characters. Exceeding this length will cause the format to be truncated."

Q: The assignment says to use semaphores to implement locks and conditions. I don't see how this would be possible since the cap on semaphores is 32. In the case that you want 33 locks, it couldn't make the 33rd one (assuming you don't have any semaphores created already). Because using SemCreate tests agains MAX_SEMS. So the only thing I can think of is to copy and paste code into the lock and condition methods and change things accordingly (MAX_LOCKS etc.). What gives?

A: In theory, there should be more semaphores for lock and cond. variable. But in this project, there are enough semaphores available for you to use, just ignore this problem.

Q: How can you tell if the caller of LockHandleAcquire is the current owner of that lock? Are we supposed to add a field to the Lock struct that keeps track of that?

A: Use currentPCB