dining-p

### Due Date: November 1, 2014 @midnight Submission Subject: “dining-p” ← this is the subject line of your email

## General Submission Criteria:

* See Lab 0 for the General Submission Criteria!
* Make a directory in your repository: lab4
* Include all of your Lab4 work within the lab4 directory

## Overview:

In this lab, you will develop a single program that behaves as a single philosopher. When this program is executed with other instances of the program, you will have up to N cooperating processes all involved in solving the “Dining Philosophers problem.”

There are two parts to the project, but you will only be turning in Part 2. Part 1 is the dining philosophers program that allows for only N-1 philosophers to dine at a table with N seats. If you were to allow another philosopher to dine, deadlock is possible.

In Part 2, you are to modify your program to remove the possibility of deadlock by removing one of the following two conditions for deadlock.

1. Hold and Wait
2. Circular Wait

## Learning Objectives:

1. To reinforce the student’s knowledge of process synchronization

2. To reinforce the student’s understanding of deadlock and how to prevent this from  
 occurring within multithreaded application

3. To expose the student to issues involved in parallel and concurrent computing

4. To expose the student to use of POSIX semaphores and to further hone their programming skills

## Executable Names:

**dining-p** : a program that simulates a single dining philosopher.   
The program is designed to be executed numerous times, simultaneously, to demonstrate the effect use of semaphores.

## Usage:

$ dining-p seats position

where

seats: the number of seats available around the circular table

position: the seat number in which the philosopher will sit

## Makefile Targets:

all: (default)

dining-p:

clean:

In your makefile, make sure you include -lpthreads (as per the man pages) during the final compilation step.

## Part I Description:

The main part of the dining philosophers program is provided in your textbook within Section 5.7.3, and in the following figure:

|  |
| --- |
| getfile.jpeg |

In your program, you should take the following steps:

1. Develop two functions: eat() and think();
   1. each function consume a random amount of real-time. (see usleep(3), rand(3))
   2. each function emits to stdout one of the following lines:
      1. Philosopher #n is eating
      2. Philosopher #n is thinking
2. Augment the code provided above:
   1. to use the eat() and think() functions
   2. to keep a count of the number of full eat-think cycles
   3. to have the wait() and signal() functions to nothing
3. Develop a signal handler for SIGTERM.  
   When your program receives this signal, it needs to
   1. effectively remove the philosopher from the eat-think cycle
   2. release any system resources (see sem\_destroy(3),  
       sem\_close(3)  
       sem\_unlink(3))
   3. emits to stderr the following line:
      1. Philosopher #n completed m cycles
   4. return success from the process
4. Add semaphores to the program (see sem\_overview(7))
   1. allocate a semaphore for each chopstick (see sem\_open(3))
   2. modify the wait() function to wait on the semaphore (see sem\_wait(3))
   3. modify the signal() function to post to the semaphore (see sem\_post(3))

* Debug your program incrementally throughout the above steps
  + You can use the “launch-philosophers” bash script to help you run your program multiple times.
  + You can use the kill command to terminate your processes
    - kill -TERM *pid*
  + *You can see that your semaphores have been created by the command*
    - *ls -l /dev/shm/*
  + If you choose to use unnamed semaphores, you can use ipcs to see the semaphores you have created
    - ipcs -s

|  |
| --- |
| $ ipcs -s  ------ Semaphore Arrays --------  key semid owner perms nsems |

* + You can use ipcrm to cleanup your semaphores
    - ipcrm -S *key*
    - ipcrm -s *semid*

## Part II Description:

In this part, you will augment your program to remove the potential of deadlock. Recall, there are 4 necessary conditions that must hold for deadlock to occur:

1. Mutual Exclusion
2. Hold and Wait
3. Non-Preemption
4. Circular Wait

Hence, to eliminate the possibility of deadlock, you need to re\*move one of these conditions from your program. The two that are viable are:

1. Hold and Wait  
   You can create an additional semaphore that is used to create a critical section around the allocation of system resources (i.e., allocation of the chopstick)
2. Circular Wait  
   You can modify the algorithm to use an asymmetric algorithm. Depending on the number of philosophers and which philosopher the current process is modeling, your algorithm would be different. For example, consider having different algorithms for left-handed and right-handed philosophers.

## Part III Description: ENHANCEMENT

*What? I thought there was only two parts?*  You are correct. This part is an optional part for Lab4. In this enhancement, you modify your current program to function like the “launch-philosophers” script.

## Executable Names:

**launch-philosophers** : a program that simulates a solution to the dining philosopher problem.

## Usage:

$ launch-philosophers seats

where

seats: the number of philosophers that are sitting at the table

## Approach:

You program should have the following general workflow.

1. create a new process group (see setpgid(2))
2. allocate the semaphores
   * You might want to use unnamed semaphores in this part
   * Note the caveat in NOTES section of the sem\_open man page
3. create <seats> processes
   * each child models a particular philosopher
4. wait for the SIGTERM to parent
5. send signal to all of your children (i.e., send it to the process group)
6. wait for all your children
7. perform any final cleanup (see sem\_unlink(2))
8. return success from the parent process