# Pass the Plate:

## A Thanksgiving Experiment with Semaphores

Rachel Klesius and Garrett Moncrief

**Abstract - in our final project we explore the concept of semaphores in a novel way: by representing the behavior of semaphore protected shared memory segments with a Thanksgiving dinner. In this program there are a set number of people who are invited to dinner (the programs) and a set number of different dishes (the shared memory segments). The program will find the number of passes each dish experiences based on which guest requests which dish. This shows multiple semaphores, processes that need a variable amount of those semaphores, and having to execute those processes in a specific order.**

### Introduction

Pass the Plate is a program the helps explore how a program responds when multiple processes are each trying to access several shared memory segments in a sequential fashion, using semaphores to protect the critical section of each process.

To explore the concept we created six semaphore protected shared memory segments and four processes which each had a list of memory segments they needed to access. The processes were forked in order and guidelines dictated which process they could get access to a shared memory segment from.

As we change the ordering of the process calls the number of times that each shared memory segment is “passed” changes. This means that given a situation where several processes are attempting to access a shared memory segment the sharing instructions and process order can make a significant difference to the time it takes for all processes to complete. Forcing an order and forcing the semaphores to go through multiple processes first once they were picked up was very fascinating to us. Any process could be the first process to pick up a resource if they were the first to need it, but then if another process in the order needed it, it would be passed from process to process until the process that needed it finally got it. It was interesting to us to be able to loosely manipulate semaphores and impact them with guidelines on who gets the resource next.

### Methodology

The program was written in C and tested in a Linux environment. The libraries included are:

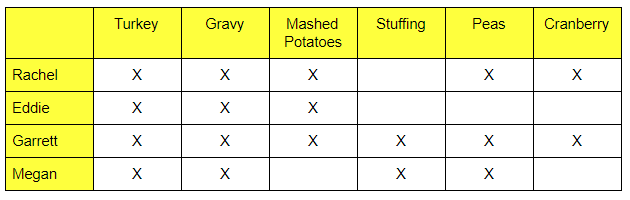
|  |
| --- |
| #include <stdio.h> |
| #include <stdbool.h> |
| #include <sys/types.h> |
| #include <sys/ipc.h> |
| #include <sys/shm.h> |
| #include <sys/sem.h> |
| #include <stdlib.h> |
| #include <unistd.h> |
| #include <errno.h> |

The first step is to define six semaphore keys and define the number of people attending the dinner. A structure is created for the shared memory segments which includes integer values for the number of passes, the current position, and how many people need the dish. POP and VOP functions are included to protect and release the critical section using semaphore.

Next, six shared memory segments need to be created and connected to for the “dishes,” for which we titled: turkey, gravy, mashed potatoes, stuffing, peas, and cranberries. For these shared memory segments we initialize the number of passes to 0, the current position to -1 (which signifies that the dish hasn’t been picked up yet), and fill in the number of people who need the dish (which is elaborated upon below).

The main function then forks each “guest” based on their seat. In our implementation these processes are titled Rachel, Eddie, Megan, and Garrett. The main function completes by waiting for the child processes to complete, printing the id of the terminating process as it exits. It then prints how many times each dish was passed before detaching and deallocating the shared memory and semaphore segments.

Each process defines what the guest wants and sets boolean values that must be satisfied before the process completes. In our implementation the guests required the following foods:



If nobody has picked a dish up or if a neighbor has a dish then the guest can protect the memory segment with a semaphore while they increment the number of times the dish has been passed, decrease the amount of need for the dish, update the location of the dish at the table, and update their boolean value to signal they’ve received the dished. After this the process releases its semaphore on the shared memory segment and prints out the current number of times the dish has been passed and who just received it.

We also need a way to “pass” the dishes along, even if the process holding the dish doesn’t need it. If there is still a need for the dish at the table, then that process has to pass it along. We identify the need for each dish with the need integer as part of the shared memory structure, which is being reduced by one when a process uses the resource it needs. If a process receives a dish and there is no more need for it, the process will not pass it, otherwise it is sent down the line with the pass count increased.

### Results

To show the difference in the amount of times each semaphore is being “passed” we included two different orderings of the dinner table. Figure 1 shows the ordering Rachel->Eddie->Megan->Garrett, and Figure 2 shows the ordering Megan->Rachel->Eddie->Garret. For Figure 1, Mashed Potatoes are passed four times, while in Figure 2 mashed potatoes are passed three times. Only three people want mashed potatoes, so the second configuration is optimized for mashed potatoes while the first is not. This is because even though Megan doesn’t want mashed potatoes, she sees that there is still a want, and the mashed potatoes have to be passed to her to get from Eddie to Garrett. Conversely, the first configuration is optimized for cranberry because only two people need it and there are exactly two passes of cranberry; where in the second configuration there is three. This is because Rachel gets the cranberry first, then it has to pass to Eddie who doesn’t want the cranberry but sees that there is still a need for it, and so he passes it to Garrett who uses the cranberry.

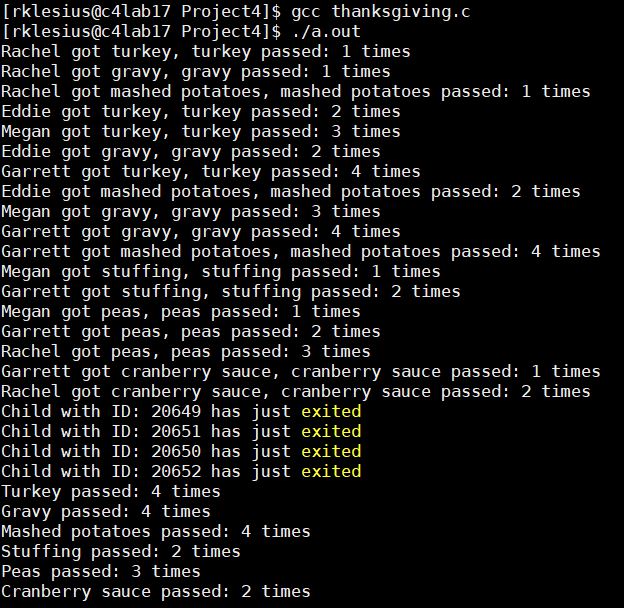
****

Figure 1

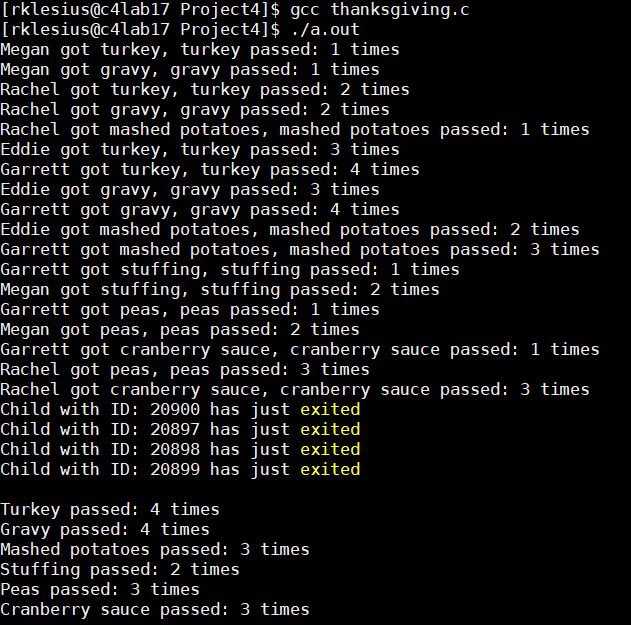
****

Figure 2

### Conclusions

Like the Dining Philosophers problem even though this isn’t a real problem it uses the concepts of resource allocation, meaning that if we can optimize this problem we can apply the concepts to real world problems. By passing plates when they are not needed or clearing them from the table after everybody is served we demonstrate deadlock avoidance. Otherwise a plate would reach a person who didn’t want that food and cause the rest of the plates to become blocked. This also prevents starvation; since the plates are constantly revolving around the table each person will eventually load their plate up with their desired food.

Since the processes are executing concurrently semaphores are utilized to ensure data consistency and to protect the critical section of each shared memory section while it is being updated. An optimal conclusion is one in which the guests are sat in a manner in which the number of passes are as close as possible to the number of people who needed the dish. This shows how, given concurrency rules, a program can be optimized simply by arranging the order in which its processes access shared memory segments.