The Cigarette Smokers Problem

Alan Rodriguez and Sara Savitz

I. Abstract

In this report, we explore the concepts of multithreading and semaphores to develop a solution to the Cigarette Smokers Problem. We discuss possible deadlocks that can occur and how to handle these deadlocks using appropriately timed semaphores. This problem demonstrates a basic task that many operating systems perform, which is allocating newly available resources to multiple processes or threads.

II. Introduction

The Cigarette Smokers Problem is a concurrency problem of computer science first conceived by Suhas Patil in 1971. It describes a scenario where there are three smokers sitting around a table. Each smoker has an infinite supply of either tobacco, papers, or matches. Another person, an agent, is also sitting at the table; he has an infinite supply of all the resources.

The agent will randomly choose two resources to place on the table. The smoker who possesses the remaining resource should collect the resources and use them to make and smoke a cigarette. This scenario continues forever, and will be implemented as such in the software.

In the real world, a deadlock would never realistically occur given this scenario. The smoker who had the third resource would always take the supplied resources and smoke the cigarette (assuming their fellow smokers aren't overly stubborn). But in the context of computing and parallel processing, the possibility of a deadlock does exist. Thus the need for semaphores occurs.

We discuss later in this report what semaphores are needed to solve the problem, as well as how thread functions will be used to simulate the actions of each person sitting at the table.

III. Methodology

The simplest approach to solving the Cigarette Smokers Problem is determining where a deadlock may occur during program execution.

For instance, imagine that the agent places tobacco and paper on the table. The smoker with paper would see the tobacco and try to take it, knowing that he could craft a cigarette with it. The smoker with tobacco would see the paper and try to do the same thing for the very same reason. Now imagine that both smokers take the resource they need, and now both smokers are unable to smoke their cigarette because no matches are available – this sequence of events has resulted in a deadlock.

We can prevent this deadlock from occurring by creating a semaphore for each person at the table. The semaphore for a particular smoker is unlocked anytime the two resources needed by that smoker are supplied by the agent. Otherwise, the smoker semaphore will wait if the resource possessed by a particular smoker is present on the table. A fourth semaphore is used to signal when the smoker has finished smoking their cigarette and it is time for the agent to supply two more resources.

Threads will be used to represent both the agent and the smokers. The thread function for the three smoker threads will be the same function, since they all will perform the same task. The agent thread will have a separate function dedicated to randomly choosing which two resources will be placed on the table next.

Execution of these two functions will be passed back and forth between the agent and the appropriate smoker thread. When one function terminates, the other function will resume, and the process will continue forever. The complementary code in the next section is the solution we used for this problem.

IV. Results

The main() function of the program is dedicated to initializing the semaphores and creating the threads as well as their functions. No shared memory segment is used in this project because all threads are contained in the same process and are accessing the same global variable.

As stated above, the smoker threads will all use the same thread function running in parallel. The smoker thread function takes in the thread ID as an argument in order to identify the thread. Using the thread ID, we can identify the proper semaphore to unlock and lock the correct smoker thread to utilize the available resources. The code for the smoker thread function is shown below:

```
void* smoker_func(void* p)
	{
		// Identify thread
		int n = (int)p;
		char* name = smokers[n];

		while(1)
		{
// Make each smoker thread wait until semaphore is signaled by agent
	switch(n) {
		case 0: sem_wait(&match); break;
		case 1: sem_wait(&paper); break;
		case 2: sem_wait(&tobacco); break;
		default:  break;
		}

			// Entering critical section
			sem_wait(&mutex);

			// Read and print resources placed
			//on table by agent thread
			int i;
			for(i=0; i<2; i++)
			{
			char c = buffer[i];

			switch (c) {
			case 'p':   printf("%c - %s removed
			paper from table.\n", c, name);
			break;
			case 'm': printf("%c - %s removed
			match from table.\n", c, name);
			break;
			case 't': printf("%c - %s removed
			tobacco from table.\n", c, name);
			break;
			default:  printf("%c - %s removed
			unknown item.\n", c, name);
				}
			}

		// Exiting critical section
		sem_post(&mutex);

		// Smoke for 1 sec
		printf("Smoker with %s smokes
		cigarette.\n", name);
		sleep(1);

		// Wake up agent thread
		sem_post(&agent);
		}
	}
```

Once the smoker function terminates, the agent semaphore will unlock, signaling the agent to choose two more resources at random to place on the table. The thread function for the agent thread is shown below:

```
void* agent_func(void* p)
	{
		// Should only be called once
		srand(time(NULL));

			while(1)
			{
			// Get a random number 0-2
			int r = rand()%3;
			printf("r = %d\n", r);

			// Entering critical section
			sem_wait(&mutex);

			switch(r) {
			case 0:
	// Agent puts Tobacco and Paper on table and signals
	smoker with matches
			buffer[0] = 'p'; buffer[1] = 't';
			printf("Agent puts Paper and
			Tobacco on table.\n");
			printf("Agent wakes smoker with
			Matches.\n");
			sem_post(&match); break;

			case 1:
	// Agent puts Matches and Tobacco on table and
	signals smoker with paper

			buffer[0] = 'm'; buffer[1] = 't';
			printf("Agent puts Match and
			Tobacco on table.\n");
			printf("Agent wakes smoker with
			Papers.\n");
			sem_post(&paper); break;

			case 2:
	// Agent puts Matches and Paper on table and signals
	smoker with tobacco

			buffer[0] = 'm'; buffer[1] = 'p';
			printf("Agent puts Match and Paper
			on table.\n");
			printf("Agent wakes smoker with
			Tobacco.\n");
			sem_post(&tobacco); break;
				}

			sem_post(&mutex);
			// Exiting critical section
			sem_wait(&agent);
			// Put agent to sleep
			}
		}
```

At the end of the agent thread function, the agent puts itself to sleep, which allows the smoker thread function to resume. The data provided by the agent function will be used to determine which smoker thread will be allocated the resources.

Note that no input is needed for the program; the agent only needs to choose two ingredients at random at the beginning of each cycle. This is handled by the rand()%3, which provides a random number in the range of 0-2.

A sample output is provided below, demonstrating all three cases with appropriate print statements:

```
r = 1
Agent puts Match and Tobacco on table.
Agent wakes smoker with Papers.
m - Paper removed match from table.
t - Paper removed tobacco from table.
Smoker with Paper smokes cigarette.
r = 0
Agent puts Paper and Tobacco on table.
Agent wakes smoker with Matches.
p - Matches removed paper from table.
t - Matches removed tobacco from table.
Smoker with Matches smokes cigarette.
r = 2
Agent puts Match and Paper on table.
Agent wakes smoker with Tobacco.
m - Tobacco removed match from table.
p - Tobacco removed paper from table.
Smoker with Tobacco smokes cigarette.
```

V. CONCLUSION

To conclude, this project provided an opportunity to learn more about semaphores and threads, as well as further demonstrate their utility and how they can be used in a variety of situations. There are multiple approaches that can be taken to solve this problem, but the approach we took was the one that minimized the number of semaphores needed. Other solutions involved having a semaphore for all people and as well as the three resources.

In addition to minimizing the number of semaphores used, we also designed our smoker thread function so it may be used by all three smoker threads. By using the thread ID to identify individual threads, we reduced the amount of code needed to implement the solution, resulting in a more concise and efficient program.