COP 4600 Operating Systems

Project 1 (Shared memory and synchronization)

```
Gregory O'Marah - net ID: gdo - U34240613
      COP 4600 Operating Systems
  Project 1 – synchronization - ass1
```

The purpose of this project was to illustrate how shared memory can be used and how parallel processes could be synchronized to update the shared memory effectively. The mechanisms studied included the Peterson's Algorithm solution of flags and turns, which was implemented to insure that only one process could update the shared memory segment at a time. Additionally, functions were implemented to create, attach, and release the shared memory properly on the system.

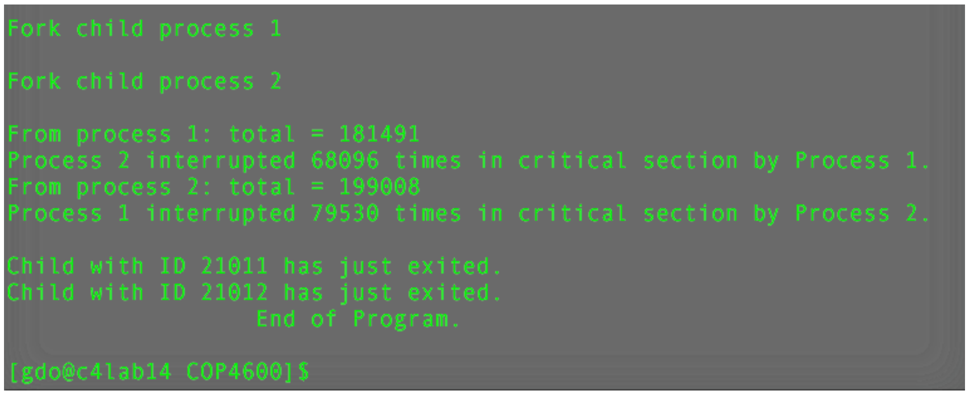
In early attempts to implement the solution, the flags and turns were initialized globally. It appears that after forking the child processes, copies of these variables were being created so that each was stored within the process1 and process2 function separately. This did not cause any interrupts as the processes were keeping separate flags and turns, and were not able to see that value as it was updated from the other process. This caused zero interrupts on all trials over 75 trials of the execution.

Next I implemented the flag and turn variables as part of the struct for shared memory (shared_mem.) This allowed the processes to update the same copy of the shared flags and turns, and to send the process into the spinlock of the waiting section, which finally proved this would work for protection of the critical section and updates to the shared memory segment. The functions also needed a separate flag to indicate if we cause the process to wait, so variables called first and second were created for this purpose. They were placed in the waiting section of the process 1 and process 2, respectively – so that we could then check them to indicate if the processes was interrupted and caused to enter the waiting section for the spinlock.

This was a successful implementation of the solution as shown in the data below (see screen shots.) We have implemented a solution that provides the three critical mechanisms of: mutual exclusion, progress, and bounded waiting. The processes were enabled to run up to 100000 for each process, process1 and process2. We can see that we have approximately 65k – 85k interrupts per each process on each execution of the program.

So in conclusion, I observed that the processes are taking turns executing the shared memory access in their critical sections, as the programs do not have much other delay in processing beyond that point. If we had a program where the process was going on to do some further I/O procedure or something not requiring the processor back so quickly, we might see fewer interrupts as we would get closer to a balance between I/O and CPU operation. As they are now, these processes are CPU bound and thus they are mostly running in their critical section or waiting in spinlock the majority of the time.

The program was successfully executed with interrupt counts at least 100 times. A sampling of the data follows in the space below:



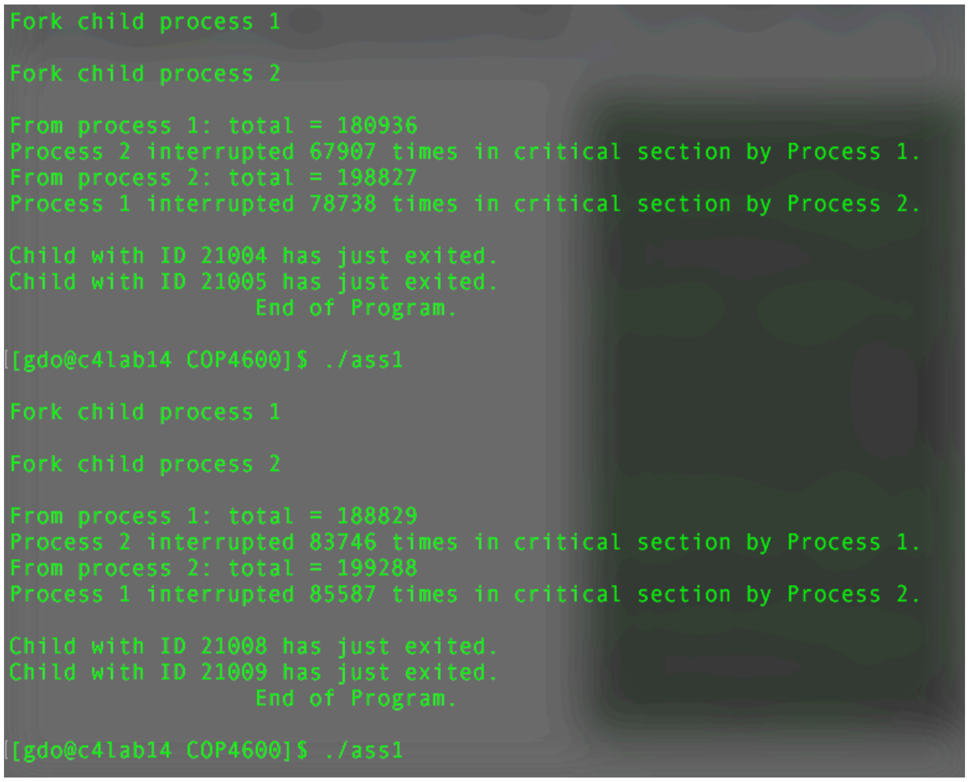
```
Fork child process 1

Fork child process 2

From process 1: total = 181491
Process 2 interrupted 68096 times in critical section by Process 1.
From process 2: total = 199008
Process 1 interrupted 79530 times in critical section by Process 2.

Child with ID 21011 has just exited.
Child with ID 21012 has just exited.
                  End of Program.

[gdo@c4lab14 COP4600]$
```



```
Fork child process 1

Fork child process 2

From process 1: total = 180936
Process 2 interrupted 67907 times in critical section by Process 1.
From process 2: total = 198827
Process 1 interrupted 78738 times in critical section by Process 2.

Child with ID 21004 has just exited.
Child with ID 21005 has just exited.
                  End of Program.

[[gdo@c4lab14 COP4600]$ ./ass1

Fork child process 1

Fork child process 2

From process 1: total = 188829
Process 2 interrupted 83746 times in critical section by Process 1.
From process 2: total = 199288
Process 1 interrupted 85587 times in critical section by Process 2.

Child with ID 21008 has just exited.
Child with ID 21009 has just exited.
                  End of Program.

[[gdo@c4lab14 COP4600]$ ./ass1
```

Program code:

```
/* Gregory O'Marah - netID: gdo
   U3424-0613
   COP 4600 Operating Systems
   project 1 - synchronization
                          */
```

```c
/*ass1*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <unistd.h>

/* change the key number */
#define SHMKEY ((key_t) 6545)

typedef struct // struct contains values for sharing between processes p1, p2
{
   int value, turn, first, second;
   int flag[2];

} shared_mem;

shared_mem *total; // declare var * (total) for shared memory between p1, p2


/*----------------------------------------------------------------------*
 * This function increases the value of shared variable "total"
 *  by one all the way to 100000
 *----------------------------------------------------------------------*/

void process1 () // p1 : process 1
{
   int k = 0, interrupt  = 0; // declare and initialize counting vars
   while (k < 100000) // k will go up to 100000
   {
      total->flag[0] = 1; // begin peterson's solution : set flag 0 = 1 (true)
      total->turn = 2;    // (process 2's turn)

      while (total->flag[1] && total->turn == 2) // *** Waiting section ***
         total->first = 1;  // first is set to indicate that P2 interrupted P1
                           // end of waiting section
                           // *****start the critical section
      if (total->second == 1)  // check to see if P1 interrupted P2
         interrupt++;         // increment counter if interrupt
      k++;
      total->value++;       // increment the value (critical shared mem access)
      total->first = 0;    // set first to 0 so we indicate not waiting
      total->flag[0] = 0; // end critical section, flag 0 set to 0 (false)
   }
   printf ("\nFrom process 1: total = %d", total->value); // p1 done, outputs to screen
   printf ("\nProcess 2 interrupted %d times in critical section by Process 1.",
interrupt);
}


/*----------------------------------------------------------------------*
 * This function increases the vlaue of shared memory variable "total"
 *  by one all the way to 100000
 *----------------------------------------------------------------------*/

void process2 () // p2 : process 2
{
   int k = 0, interrupt  = 0; // declare and initialize counting vars
   while (k < 100000) // k will go up to 100000
   {
      total->flag[1] = 1;// begin peterson's solution : set flag 1 = 1 (true)
      total->turn = 1;  // (process 1's turn)

      while (total->flag[0] && total->turn == 1) // *** Waiting Section ***
         total->second = 1; // second set to indicate that P1 interrupted P2
                           // end of waiting section
                           // *****start the critical section

      if (total->first == 1) // check to see if P2 interrupted P1
         interrupt++;         // increment counter if interrupt

      k++;
      total->value++;       // increment the value (critical shared mem access)
      total->second = 0;    // set first to 0 so we indicate not waiting
      total->flag[1] = 0;   // end critical section, flag 1 set to 0 (false)
   }
   printf ("\nFrom process 2: total = %d", total->value);   // p2 finished, output
   printf ("\nProcess 1 interrupted %d times in critical section by Process 2.",
interrupt);
}



/*----------------------------------------------------------------------*
 * MAIN()
 *----------------------------------------------------------------------*/

int main()
{

   int   shmid;
   int   pid1;
   int   pid2;
   int   ID;
   int  status;
   char *shmadd;
   shmadd = (char *) 0;

   /* Create and connect to a shared memory segmentt*/

   if ((shmid = shmget (SHMKEY, sizeof(int), IPC_CREAT | 0666)) < 0)
   {
      perror ("shmget");
      exit (1);
   }

   // attach total to shared mem segment
   if ((total = (shared_mem *) shmat (shmid, shmadd, 0)) == (shared_mem *) -1)
   {
      perror ("shmat");
      exit (0);
   }

   total->value = 0; // initialize total-value to zero

   if ((pid1 = fork()) == 0)
   {
      printf ("\nFork child process 1\n"); //  fork child 1, process 1
      process1();
   }
   if ((pid1 != 0) && (pid2 = fork()) == 0) // fork child 2, process 2
   {
      printf ("\nFork child process 2\n");
      process2();
   }

   if ((pid1 != 0) && (pid2 != 0))
   {
      // get pids and wait for children to exit
      int finished = wait(&status);
      ID = finished;
      finished = wait(&status);
      // print out child status on exit
      printf("\n\nChild with ID %d has just exited.\r\n", (pid1 == finished)?pid1:ID );
      printf("Child with ID %d has just exited.\r\n", (pid2 == ID)?pid2:finished );

      // free shared memory
      if ((shmctl (shmid, IPC_RMID, (struct shmid_ds *) 0)) == -1)
      {
         perror ("shmctl");
         exit (-1);
      }

      // end of program, termination
      printf ("\t\t  End of Program.\n\n");
   }
   return 0;

}
```