CS39002 Operating Systems Laboratory
Spring 2025

Lab Assignment: 5
Date: 05–February –2025

IPC using shared memory

In this assignment, one leader process L and n follower processes $F_1, F_2, \ldots, F_n$ cooperatively share a memory segment M. The leader L first creates the segment M. The individual int-valued cells of M store the following items.

- $M[0]$ always stores n (the number of followers).
- $M[1]$ stores the number of followers that have joined so far.
- $M[2]$ stores the turn of the process to access the shared memory (0 for leader L, i for Follower F_i).
- $M[3]$ is the cell where the leader L writes.
- $M[4], M[5], \ldots, M[3+n]$ are the cells where the followers $F_1, F_2, \ldots, F_n$ respectively write.

Assume that $n \leqslant 100$. Based upon this limit, determine the maximum size of the segment M.

After creating M, the leader L waits until all of the n followers join. Race condition is possible during the joining of the followers, because they access and modify $M[1]$. In reality, a situation where race conditions may leave $M[1]$ in an inconsistent state is very rare. Ignore such rare occurrences, that is, use no preventive measures to ensure mutual exclusion of $M[1]$. If it is a problem on your platform, use *sleep*() judiciously.

Subsequently, no race condition is possible. This is ensured by alternating the turns of the leader and the followers, as indicated by $M[2]$. Each process makes a busy wait until its turn comes. When it is done with its turn, the process sets $M[2]$ to break the busy wait of the next process. The processes enter the following loop. L initializes $M[2]$ to 0 (so it is the turn of the leader first).

> L writes a random integer in the range 1 – 99 to $M[3]$.
>
> L then sets $M[2]$ to 1 breaking the wait of F_1, and itself waits (busy) until $M[2]$ becomes 0 again.
>
> F_1 writes a random integer in the range 1 – 9 to $M[4]$, sets $M[2] = 2$, and goes to a busy wait until $M[2]$ becomes 1 again.
>
> F_2 writes a random integer in the range 1 – 9 to $M[5]$, sets $M[2] = 3$, and goes to a busy wait until $M[2]$ becomes 2 again.
>
> F_n writes a random integer in the range 1 – 9 to $M[3+n]$, sets $M[2] = 0$, and goes to a busy wait until $M[2]$ becomes n again.
>
> L wakes up from its busy wait, adds $M[3], M[4], M[5], \ldots, M[3+n]$, and prints the sum.

In order that the above loop terminates, L keeps track of the sums it has computed so far. It uses an array as a hash table to remember the sums generated (do this yourself instead of using a ready-made hash-table implementation). This array is not shared, but is privately stored in L's local memory. As soon as a sum is duplicated (for the first time), L sets $M[2] = –1$ (instead of 1). The turn of each follower F_i comes when $M[2]$ becomes i or $–i$. The behavior of F_i when it sees $M[2] = i$ is already explained in the above loop. When F_i sees $M[2] = – i$, it terminates, but before doing so, F_i sets $M[2]$ to $–(i + 1)$ or 0, in order to terminate the busy wait of F_{i+1} (if $i < n$) or of L (if $i = n$). When L wakes up for the last time, it understands that all followers are now gone, so L removes M, and itself terminates.

Code for the leader *L*

Write *leader.c*(*pp*) to carry out the work of *L*. You may supply *n* as a command-line argument. In absence of that argument, take the default value $n = 10$. The program first creates the shared-memory segment *M*, initializes *M*[0], *M*[1], and *M*[2], and waits until *M*[1] becomes *n*. After this wait is over (that is, all of the *n* followers have joined), *L* enters the sum-computation loop with the followers. Upon the termination of the loop (a sum is repeated), *L* engages in a final round of conversation with the followers. Finally, *L* removes *M*, and exits.

Code for the followers

Write *follower.c*(*pp*) for this purpose. Its compiled version would run from multiple shells (terminal windows). A command-line argument specifies how many followers, call it n_f, this run will generate. The default value of n_f to be used (in absence of the command-line argument) is 1. After n_f is determined, n_f child processes are created, each playing the role of a follower. A follower first gets the ID of the shared-memory segment *M* (it must not create it). It then gets its follower number *i* after incrementing *M*[1]. Subsequently, it joins the summand-generation loop, and finally takes part in the termination round, as described earlier. When done, each follower exits (*M* is not removed by any follower). Its parent waits for all its follower children to terminate. Note that in this assignment, the followers are not forked by the leader process. Instead, the followers are children of independent runs of processes that run the executable from *follower.c*.

Notes

- The leader and the followers are launched by separate programs, and are not in a child-parent relationship. They should use *ftok*() to agree upon the ID of the shared-memory segment *M*. Supply a common directory (like /) as the first argument to *ftok*(), because your code must run on TAs' platforms.

- At any point of time, at most one instance of the leader should be running. An attempt to run a second leader concurrently must fail. Use IPC_EXCL during the creation of *M* in order to achieve this.

- You run the executable from *follower.c* from multiple windows, possibly with various command-line arguments. If the total number of followers exceeds *n*, attempts to create F_i with $i > n$ must fail. That is, each child process in *follower.c* must exit gracefully as soon as it sees that *M*[1] has already reached *n* (*n* is stored in *M*[0]).

- A follower does not create *M*. An attempt to run follower without the leader already running should fail. This can be detected when the follower tries to get the ID of *M*. The follower does not create *M*, so an attempt to get the ID of *M* would fail in the absence of the leader.

Submit only the two files *leader.c*(*pp*) and *follower.c*(*pp*).

Sample Output

Terminal 1

```
$ ./leader 15
```

Wait for the moment.

Terminal 2

```
$ ./follower
follower 1 joins
```

Terminal 3

```
$ ./follower 10
follower 2 joins
follower 3 joins
follower 4 joins
follower 5 joins
follower 6 joins
follower 7 joins
follower 8 joins
follower 9 joins
follower 10 joins
follower 11 joins
```

Terminal 4

```
$ ./follower 8
follower 12 joins
follower 13 joins
follower error: 15 followers have already joined
follower 14 joins
follower 15 joins
follower error: 15 followers have already joined
follower error: 15 followers have already joined
follower error: 15 followers have already joined
```

Terminal 1 (where leader is running) now shows the following output

```
88 + 4 + 9 + 4 + 8 + 8 + 4 + 3 + 9 + 7 + 5 + 3 + 4 + 6 + 1 + 6 = 169
25 + 9 + 7 + 4 + 1 + 8 + 4 + 7 + 8 + 5 + 4 + 5 + 5 + 8 + 4 + 9 = 113
34 + 9 + 9 + 4 + 5 + 1 + 6 + 4 + 5 + 8 + 6 + 4 + 5 + 8 + 6 + 9 = 123
38 + 5 + 7 + 2 + 4 + 1 + 8 + 8 + 9 + 3 + 1 + 8 + 9 + 1 + 9 + 4 = 117
 7 + 9 + 2 + 1 + 7 + 5 + 8 + 8 + 9 + 9 + 1 + 2 + 2 + 2 + 9 + 3 =  84
23 + 3 + 5 + 2 + 8 + 4 + 4 + 7 + 4 + 4 + 3 + 7 + 1 + 9 + 1 + 8 =  93
19 + 4 + 9 + 2 + 4 + 3 + 9 + 8 + 5 + 8 + 2 + 6 + 8 + 6 + 7 + 9 = 109
28 + 9 + 4 + 2 + 6 + 6 + 9 + 8 + 7 + 7 + 6 + 3 + 5 + 9 + 3 + 8 = 120
78 + 5 + 1 + 5 + 9 + 8 + 1 + 1 + 5 + 3 + 7 + 7 + 8 + 6 + 2 + 9 = 155
45 + 1 + 5 + 6 + 2 + 2 + 5 + 7 + 8 + 5 + 5 + 7 + 6 + 6 + 2 + 5 = 117
$
```

The follower terminals show the following outputs.

Terminal 2

```
follower 1 leaves
$
```

Terminal 3

```
follower 3 leaves
follower 4 leaves
follower 5 leaves
follower 2 leaves
follower 7 leaves
follower 9 leaves
follower 11 leaves
follower 8 leaves
follower 6 leaves
follower 10 leaves
$
```

Terminal 4

```
follower 13 leaves
follower 14 leaves
follower 15 leaves
follower 12 leaves
$
```