CS240 ASSIGNMENT 4:

Semaphore

NAME: XIAOPENG XU KAUST ID: 129052

Abstract:

In this assignment I added semaphore functionality to the original xv6 code. I added semaphore structures in the system. 4 system calls for semaphore operation are implemented, and they're tested using producer and consumer example.

Design & Implementation:

Below are the design decisions I made:

1. Semaphore structure and contents:
   ```
   uint name;      // name of semaphore
   int value;      // value of current semaphore
   int flag;   // semaphore EMPTY/FULL
   int proc_nr;// processes using this semaphore
   struct spinlock lock; //lock of semaphore
   ```
   Flag EMPTY means the semaphore is not used currently, FULL means it is used. The semaphore structure `sem_t` was defined in *sem.h* along with several macros.
2. Semaphores in the system are managed by OS through a semaphore table `semtab`. This `semtab` was defined in *sem.h*.
3. Semaphore in a processes in stored in process through an array `proc->semhd[]`. `Proc->semhd[]`stores the handles of semaphore used in the process. When semhd[i] is empty, `proc->semhd[i]` equals to `0`. This is implemented in file *proc.h*.
4. In implementation, 4 system calls are added for semaphore operations. These system calls include `sem_get`, `sem_delete`, `sem_signal`, and `sem_wait`.
   a. `Sem_get` get a semaphore handle using semaphore name and value. First it searches in existing semaphore to find semaphore with the same `name`, and return `semhd` if found. If the semaphore does not exist, `sem_get` initiates a new semaphore and returns the `semhd`. It returns `-1` if maximum number of semaphores in process – `PROC_SEM_MAX` is reached and `OUT_OF_SEM`, i.e. `-2`, if maximum number of semaphores in the OS – `NSEM` is reached.
   b. `Sem_delete` delete a semaphore from current process. If current process is the only process using that semaphore, that is `proc_nr` equals to 1, the semaphore will be cleared from system. It returns `SEM_OK`, i.e. `0`, if semaphore was successfully deleted, `-1` if current process does not contain this semaphore.
   c. `Sem_signal` acquires the semaphore `lock` and increases semaphore `value` by one. After that, if the `value` greater than `0`, it wakes up the process sleeps with the address of semaphore `value` by call `wakeup`. After that, It releases the `lock` and return `SEM_OK`.

   d. `Sem_wait` acquires the semaphore `lock`. If the `value` of semaphore is smaller than or equal to `0`, it sleeps with the address of semaphore `value` and the semaphore `lock` until the semaphore `value` was greater than 0. Then it decreases semaphore `value` by one and releases the semaphore `lock`.

   Files modified for of theses system calls include *syscall.c*, *syscall.h*, *user.h*, *sem.c*, and *usys.S*.

5. In support of above system calls, `findsem` function is implemented in file *sem.c* to find the index of a semaphore in array proc->semhd[] if process contains this semaphore, or return a new index if the semaphore is new to this process. It returns -1 if the maximum number of semaphores in a process -- `PROC_SEM_MAX` is reached.
6. `Fork` was modified to copy the semaphores from parent to child process in file *proc.c*.
7. To test semaphore, I used the producer and consumer example. The KSM implemented in assignment 3 was used for producer and consumer to write to the same region and previous implementation of a lock for `ksmtable` – `ksmtable.lock` was removed for this test. In my semaphore test, there're 1 producer and 2 consumers which read and write to a 10 byte shared `buffer`. 3 semaphores are used, `semEmpty`, `semFull`, and `semMutex`. `SemEmpty` records the number of `'E'` words in the `buffer`, and is initiated with 10. `SemFull` records how many `'F'` words in the `buffer`, and is initiated with 0. `SemMutex` makes sure that only one process is modifying the `buffer`, and is initiated with 1.
   a. Each time the producer waits until `semEmpty` greater than 0, acquires the `Mutex` lock, writes to `buffer`, release the `Mutex` lock, and finally `signals` the consumer, which is waiting for `semFull`.
   b. Each time the consumer waits until `semFull` greater than 0, acquires the `Mutex` lock, clears a byte of `buffer`, release the `Mutex` lock, and finally `signals` the consumer, which is waiting for `semEmpty`.

Results:

Successfully passed `semtest` and `usertests`.