CS330_Assignment:3

Group 02

October 2022

1 Implementation Details

Condition Variable Implementation: The condition variable *cv* is essentially an *int* variable defined in *kernel/condvar.h*. The function *cond_wait* is implemented by calling *condsleep* function in *kernel/proc.c* which acquired the process lock, then releases the sleeplock sent along with the condition variable. This process of acquiring the process lock then releasing the sleep lock has been made atomic by the use of a dummy sleeplock because if two process on two different CPUs reach the stage of acquiring their locks simultaneously then it would cause a deadlock beacuse when they try to release their corresponding sleeplock, it would call the wakeup function, which tries to acquire the process locks of every process other than the current process. This would give rise to a deadlock.
The rest of the condsleep function is same as the original sleep function except that it puts the condition variable on the chan.
The *cond_signal* function has been implemented using the *wakeupone* function in *kernel/proc.c*, the implementation of which is exactly same as described in the assignment itself and the condition variable is sent as the chan variable.
The *cond_broadcast* function has been implemented using the *wakeup* function in *kernel/proc.c* by sending the condition variable as the chan variable.

Semaphore Implementation: The semaphore or *sem_t* structure is declared in the *kernel/semaphore.h* file and the implementation of the *sem_init, sem_wait* and *sem_post* functions is contained in the *kernel/semaphore.c* file.
sem_t consists of an integer *value*, a condition variable *cv* and a sleeplock *lock*. The implementation for these functions is exactly the same as the corresponding functions for *zem_t* defined in the lectures.

Syscalls:
Appropriate function declarations have been added to the required files such as *user.h, defs.h, user.pl, ... etc* with the function logic being contained in *sysproc.c*

1. **barrier_alloc**: A new struct *barr* has been defined in *kernel/barr.h*. The same file also contains the array of barriers of data type *barr*. This system

call check the *set* field of each element of the barrier array and as soon as it finds one with value equal to the default value (which implies it is not in use), it returns the index of that element.

2. **barrier**: This system call acquires the lock of barrier element with id equal to the one sent as the parameter. then it prints the "entered statement" by acquiring and then releasing a global sleeplock. Then it increments the number of processes in the barrier by one. If the number of processes is less than the required limit, it calls the *cond_wait* function on the condition variable and lock of that barrier, otherwise calls the *cond_broadcast* function on the condition variable of that barrier and resets the process count to 0. Then it prints the "Finished" statements again by acquiring and releasing the global sleeplock and then releases the lock of the barrier as well.

3. **barrier_free**: This system call checks if the barrier with the passed id is set or not. If it is set, it unsets it by making the set field of the barrier 0.

4. **buffer_sem_init**: The corresponding function in *sysproc.c*, *sys_buffer_sem_init*, calls the *sembufferinit* function defined in the *kernel/proc.c* file. This initializes the semaphores *pro, con, empty, full* to 1, 1, *SEM_BUFFER_SIZE*, 0 respectively and initializes the buffer values to -1.

5. **sem_produce**: The corresponding function in *sysproc.c*, *sys_sem_produce*, waits on the *empty* semaphore, acquires the *pro* lock, writes to index pointed to by the *tail* pointer and moves the tail pointer to the next location, releases the *pro* lock, does a post on the *full* semaphore.

6. **sem_consume** The corresponding function in *sysproc.c*, *sys_sem_consume*, waits on the *full* semaphore, acquires the *con* lock, writes to index pointed to by the *head* pointer and moves the head pointer to the next location, releases the *con* lock, does a post on the *empty* semaphore.

7. **buffer_cond_init** : In the *sys_proc.c* we have initialized the locks. We have also initialized every buffer element to -1

8. **cond_produce** : The function for the conditional variable producer is written in the *proc.c* file.

9. **cond_consume** : The function for the conditional variable consumer is written in the *proc.c* file.

For the last two calls, *cond_consume* and *cond_produce* the implementation is like the one that was shown in class.

2 Time benchmarks

- Argument : 70 10 10 — Conditional variable : 24 — Semaphore : 38

- Argument : 20 20 10 — Conditional variable : 16 — Semaphore : 24
- Argument : 30 10 4 — Conditional variable : 10 — Semaphore : 15
- Argument : 45 5 10 — Conditional variable : 11 — Semaphore : 13
- Argument : 50 8 10 — Conditional variable : 16 — Semaphore : 22

We observe that the conditioned variables are consistently faster. The reason is as follows:

In the semaphore program, the entire code for the producer and the consumer is between the *sem-wait* and *sem-post* calls for a binary semaphore. This means that there is no concurrency between producers. There is also no concurrency between consumers. But concurrent execution of a producer and consumer is allowed.

In the conditioned variable case, the only part between the conditioned variable locks is the part where the index is acquired. Thus not the entire work of the producer or consumer is done in a locked state. Thus there is some concurrency between producers. There is also some concurrency between consumers.

Since we have 2 cores allocated, more concurrency reduces execution time.