CS330 Assignment 3

Shubhan R	Manas Gupta	Shivam Malhotra	Dishay Mehta
200971	200554	190808	200341

November 15, 2022

1 Implementation of Conditional Variable

The **struct cond_t** has only a **spinlock yo**. This is a dummy lock that will be used later in *condsleep()* of **proc.c**. The reason a spinlock was chosen is because it directly disables timer interrupts. *cond_init()* just initializes the spinlock.

void cond_wait (cond_t *cv, struct sleeplock *lock): *cond_wait()* of **condvar.c** just calls *condsleep()* of **proc.c**. *condsleep()* is implemented in a similar manner to how sleep is implemented. In **condsleep()**, a **sleeplock lock** is passed instead of spinlock. There was a deadlock of spinlocks and the scheduler stopped working in a few cases when process lock was acquired before releasing the passed **lock**(which is how it was done in sleep). Switching the order of the statements removed the deadlock but there was a possibility that one of the threads might sleep forever if a context switch happened just after releasing the passed lock and before acquiring the process lock. So, to fix this issue, both the instructions had to execute atomically. So, I acquired the dummy spinlock yo of the condition variable, execute both statements and release the dummy lock yo of the condition variable. This is a very small code segment within spinlock (similar to how sleeplock was implemented). The **chan** on which the process sleeps is the passed condition variable itself.

cond_signal(cond_t *cv): This just calls *wakeupone()* of **proc.c**, which wakes up one of the processes sleeping on the condition variable **cv** and exits when first such process is found.

cond_broadcast(cond_t *cv): This just calls *wakeup()* of **proc.c**, which wakes up all of the processes sleeping on the condition variable **cv**.

2 Implementation of Semaphores

The implementation is exactly same as explained in slides. For semaphores, we declared a struct **semaphore** with 3 fields: curr (int), cv (struct cond_t) and lk (struct sleeplock).

We created a **sem_init** function which initializes a semaphore by accepting a semaphore struct and a integer value as argument and sets the parameter curr of semaphore as that integer and then calls *initcondvariable* and *initsleeplock* over the semaphore condition variable and sleeplock respectively.

sem_wait() and **sem_post()** are implemented exactly as given in slides

3 Implementation of System Calls

- int **barrier_alloc**(): Our barrier array is made up of **struct barrierelement**. Each **struct barrierelement** has the following variables: a sleeplock **s** for the whole element, a sleeplock **printlock** to be used while printing values, an integer flag **alloted** to say whether this is **struct barrierelement** is currently being used, an integer **n_finished** which keeps track of how many threads have completed the current instance. An integer **ins** to keep track of current instance. A condition variable **cv** to sleep on if instance is completed by a thread. An integer flag **setinst** which is used to set **ins** to the first ever instance number passed by any thread(should be 0). The barrier array **bar** is in proc.c and when the system boots up, all the values of the array are initialized using *initbarrierelement()* function.
  This syscall is used to allocate a barrier. The function runs a loop and checks if there is any available barrier and if available, changes it to alloted, and returns the first such index it encounters.

- void **barrier**(**int** ins, **int** id, **int** nproc) : We first print that the thread entered the barrier while holding **printlock** and then release the lock.Then we hold the **lock s** of the barrier. The first ever time **barrier()** is called, the value **ins** of barrier is set to the initial instance number passed. This is not done for future calls. Now, we increment **n_finished**. We now check for the condition if **n_finished == nproc**. If it is true, this thread which is the last to run the instance wakes up all the sleeping threads sleeping on **cv**, resets **n_finished** to 0, and increments **ins** of the barrier by 1. Next, all threads check condition of a while loop, which loops on the condition to see if the current instance that should be run (**ins** stored in barrier element) is same as the the function argument **ins** passed. If this is true, it just goes to sleep on the condition variable **cv**. All the threads except the last thread to run on the instance sleep inside the while loop, while the last thread enters the if condition described above and increments **ins** of the barrier. So, this thread won't enter the while loop, and all the other woken up threads also exit the while loop and release the **sleeplock s** of the barrier they held before. Then they all print that they exited the barrier after acquiring printlock separately and then they release printlock. All of them then enter the next instance together.

- void **barrier_free**(**int** i): This syscall is used to un-allocate a barrier and free it. We acquire the sleeplock for the respective i^{th} barrier and then reset all of its values and finish execution after releasing the lock hence exiting the critical section.

- void **buffer_cond_init**(): Our buffer array is made up of **struct buffereelemcond** variables. **struct buffereelemcond** has a sleeplock **s**, 2 condition variables **empty** and **full**, the integer **val** to hold value and an integer flag **isfull** to indicate if it has been filled. We iterate over all elements of the buffer array here, and for each element, the condition variables are initialized using *initcondvariable()*, the integers are initialized to 0 and the sleeplock is initialized using *initsleeplock()*.

- void **cond_produce**(**int**): Exactly done using same idea from producer() function of **multi_prod_multi_cons.c** using our own variables.

- void **cond_consume**(): Exactly done using same idea from consumer() function of **multi_prod_multi_cons.c** using our own variables. In our code, once we get the value, it is immediately printed within the same locks itself.

- void **buffer_sem_init**(): Our buffer array is made up of just integers. There are 4 global semaphores: pro and con (both initialized to 1) and empty (initialized to BUFFERSIZE) and full(initialized to 0). There are also 2 global integers nextp and nextc (which are initialized to 0). All these initialized in this function.

- void **sem_produce**(**int**): This is done exactly using idea and code given in slides.

- void **sem_consume**(): This is done exactly using idea and code given in slides. In my code, once we get the value, it is printed immediately.

4 Observations

- Running condprodconstest 100 3 2
  Start time: 64 End time: 65
  Running semprodconstest 100 3 2
  Start time: 174 End time: 176
  We can see that the time difference is 1 and 2 respectively.

- Running condprodconstest 1000 3 2
  Start time: 328 End time: 337
  Running semprodconstest 1000 3 2
  Start time: 480 End time: 495
  We can see that the time difference is 9 and 15 respectively.

- Running condprodconstest 10000 3 2
  Start time: 81 End time: 179
  Running semprodconstest 10000 3 2
  Start time: 826 End time: 1004
  We can see that the time difference is 98 and 178 respectively.

We observe that semaphore implementation takes more time than condition variable implementation. This is because in semaphore implementation, there is very little concurrency among the set of producers and among the set of consumers. Of course, there is concurrency across the producers and consumers. Only one producer can insert at a time and only one consumer can consume at a time here. But in condition variable implementation, there is concurrency among producers and consumers as well and many producers can insert at different locations at the same time and many consumers can consume from the different locations at the same time.