CS330
Assignment - 3
Team Name:

amogOS

Soham Samaddar 200990

Aditya Tanwar 200057

Samarth Arora 200849

Sarthak Kohli 200886

November 2022

1 Condition Variable

The `condition variable` struct is defined in the `condvar.h` file.

Initially, the `struct` for a condition variable was kept empty since we only wanted to use its pointer as the channel to be slept upon. Later, we added some variables to ensure that the addresses of different condition variables are necessarily different (since they now occupy non-zero amount of memory). These members are not necessarily superfluous, they could be used to store the condition variable's name, and/or the process which sends the signal/broadcasts.

The functions associated with `condition variables` (referred to as `cv` from now on) are defined in `condvar.c` and `proc.c` and their implementations are discussed briefly below:

> `wakeupone:` Written in `proc.c`. The implementation is similar to that of `wakeup` in the same file, except that as soon as a process sleeping on the received channel is woken up, the function breaks out of the `for` loop.
>
> `condsleep:` Written in `proc.c`. The implementation is similar to that of `sleep` except that it receives a `cv` in place of a channel, and makes the process sleep on this `cv` instead.
>
> `cond_wait:` Written in `condvar.c`. The function first ensures that the `sleeplock` is indeed held by the process in the first place, since it is released in `condsleep()`. Afterwards, it simply calls `condsleep()`.
>
> `cond_broadcast:` Written in `condvar.c`. The function receives the pointer to a `cv` and calls `wakeup()` to wake up <u>all</u> processes which were sleeping on the (received) `cv`.
>
> `cond_signal:` Written in `condvar.c`. The function receives the pointer to a `cv` and calls `wakeupone()` to wake up at max a <u>single</u> process which was sleeping on the (received) `cv`.

2 Semaphore

The `semaphore` struct is defined in the `semaphore.h` file. It contains the semaphore value, a sleeplock and a condition variable. The functions associated with semaphores are defined in `semaphore.c`. They are:

> `sem_init:` Initializes the semaphore with the given value and also initializes the sleeplock by calling `initsleeplock`.
>
> `sem_wait`: It checks if the semaphore value is 0. If yes, it waits (inside a while loop) on the condition variable of the semaphore till the value becomes positive. Once the semaphore value becomes positive, it decrements it (since the current process now holds one resource). All of this is guarded by the semaphore lock since this is a critical section.
>
> `sem_post`: It increases the semaphore value and signals the same via the condition variable. Again, all of this guarded by the semaphore lock since increments are critical.

3 Barrier Testing

`barrier.h` contains the `NUM_BARRIERS` constant initialized to 10. The `barrier` struct is also defined here. It contains:

`int count`: The number of processes waiting on the barrier

`sleeplock lk`: Safeguards barrier critical sections

`cond_t cv`: Condition variable for signalling when all processes reach the barrier

`int pid`: Stores the pid of the process which **initialized** this barrier. Used for checking if barrier is currently being used or not.

The barrier array named `barriers` is defined and declared in `proc.c`. We now define the syscalls related to barriers.

`barrier_alloc`: Before iterating through all the barrier slots in the barrier array, a `barrier_lock` is acquired to make it critical. A free barrier slot is checked from the pid stored in the `barrier` struct. Once a free slot is found, the count is initialized to 0, the pid value is updated with the current process pid , the sleeplock of the barrier is initialized using `initsleeplock` and the `barrier_lock` is released. Otherwise, if no free slot is found, we loop again (using the outer infinite loop).

`barrier`: First, it acquires the barrier elements' lock. Then it prints the necessary message. Then it increments the number of processes that has entered the barrier. If this count is less than the total number of processes which have to enter the barrier, it sleeps on the condition variable. Otherwise, if this is the final process entering the barrier, it resets the barrier count and `broadcasts` all the processes waiting on that condition variable. The processes print an appropriate finishing message and release the barrier elements' lock. All the print statements are safeguarded with a `print_lock`.

`barrier_free`: Acquires the `barrier_lock` and frees the given index by resetting the count to 0 and the pid to 0. Then the `barrier_lock` is released.

4 Condition Variable Buffer Testing

The bounded buffer problem using condition variables uses a `cond_buffer` struct defined in `buffer.h`. It contains:

`int x`: The value produced by the producer

`int full`: A flag for checking if the current element is full

`sleeplock lk`: Lock to safeguard from concurrent updates to buffer elements

`cond_t inserted`: Condition variable to signal that an element has been inserted from that buffer element

`cond_t deleted`: Condition variable to signal that an element has been deleted from that buffer element `buffer.h` also contains the `NUM_COND_BUFFERS` constant. The `cond_buffers` array is declared and defined in `proc.c`. Two sleeplocks `cond_bufferinsert_lock` and `cond_bufferdelete_lock` are also defined here. We now define the syscalls related to conditional variable implementation of buffers.

> `buffer_cond_init`: The head and tail variables of the buffer are initialized to 0. The insert and delete sleeplocks of the buffer array are initialized. Then we iterate through all the buffer elements and set `x` to a dummy value of -1, set their `full` flag to false and initialize their sleeplocks.
>
> `cond_produce`: The value to be produced is passed through the function in `prod`. First, we acquire the `cond_bufferinsert_lock` to obtain an index where an insertion is to be done. The tail of the buffer is shifted accordingly and the `cond_bufferinsert_lock` is released. Then, we acquire the lock of the index where we will produce. While we find that the index already has an element, we wait for the index to be free by the `deleted` condition variable signal. Once the index is free, we produce the value, set the `full` flag to true, signal the `inserted` condition variable and release the element lock.
>
> `cond_consume`: First, we acquire the `cond_bufferdelete_lock` to obtain the index where a deletion is to be made. Update the head pointer accordingly and release the lock. Then, we acquire the lock of the index where we will consume. While we find that the index is empty, we wait for the index to be full by the `inserted` condition variable signal. Once the index is full, we consume the value, set the `full` flag to false, signal the `deleted` condition variable and release the element lock. Finally, return the consumed value.

5 Semaphore MP-MC BB Testing

Since our implementation was heavily inspired from the one in lecture slides, `sem_buffer` was `typedef`-ed to `int`, since it does not require any overhead (`sleeplock`, `cv`, etc.). This `typedef` is written in `buffer.h`.

Moving ahead, some auxiliary variables were created for the (semaphore) buffer array. All of these are initialized when `buffer_sem_init` is called. The variables have been elaborated below:

> `bin_sem_pro:` This is a binary semaphore which must be held by a producer before producing an item.
>
> `bin_sem_con:` This is a binary semaphore which must be held by a consumer before picking an item from the array.
>
> `sem_buffer_head:` The production of items takes place at this index in the array. It can only be accessed in a producer function (except at initialization, when it is initialized to 0) and the `bin_sem_pro` semaphore needs to be "grabbed" before this variable can be accessed. The variable wraps around the array.
>
> `sem_buffer_tail:` The consumption of items takes place at this index in the array. It can only be accessed in a consumer function (except at initialization, when it is initialized to 0) and the `bin_sem_con` semaphore needs to be "grabbed" before this variable can be accessed. The variable wraps around the array similar to `bin_sem_head`.

`sem_full`: This semaphore is initialized to 0, and represents the number of items that are ready to be consumed.

`sem_empty`: This semaphore is initialized to the number of items possible, and represents the number of item slots that are empty.

Lastly, the functions used for implementing (sem) MP-MC BB are elaborated below:

`buffer_sem_init`: This function is responsible for initializing all the auxiliary variables stated above as well as the `sem_buffer` array. Each index in the array is initialized to `-1` (assuming that no item equal to `-1` is ever produced).

`sem_produce`: Takes as input the produced item and puts it into the array. First, it checks if there is an empty place by grabbing (`''wait''`-ing for) the semaphore `sem_empty`, then grabs the semaphore for producing (`bin_sem_pro`), and puts the item, followed by updating `sem_buffer_head`. Lastly, it `''posts''` the semaphores `bin_sem_pro` and `sem_full`.

`sem_consume`: This function takes no input. First, it checks if there is an available item by grabbing (`''wait''`-ing for) the semaphore `sem_full`, then grabs the semaphore for consuming (`bin_sem_con`), and picks the item, removes it from the array, and updates `sem_buffer_tail`. It then `''posts''` the semaphores `bin_sem_con` and `sem_empty`. Lastly, it acquires the `print_lock`, consumes the item (by printing it) and returns the item so consumed.

6 Condition Variable v/s Semaphore Implementation Comparison

Input	`Condition Variable`	`Semaphore`
20 3 2	2	3
70 3 10	7	10
1000 1 1	27	44
1000 5 10	144	235

In all test cases, the condition variable outperforms the semaphore implementation. This is to be expected since the implementation using condition variables allows for far better concurrency than the implementation using semaphores. In the semaphore implementation, at any point of time, only one producer is able to produce in the buffer and only one consumer is able to consume in the buffer. This is not the case with the condition variable implementation.

Even in the situation when there is no concurrency involved (1 producer, 1 consumer), the condition variable still outperforms semaphores. This arises because `sem_post` has an extra overhead involved with acquiring and releasing the semaphore lock, while there is no such overhead with the condition variable implementation.

7 The `releasecondsleep_lock`

This is a lock used in the implementation of the `condsleep` function in `proc.c`. The `condsleep` function is identical to the `sleep` function except that it takes a `cond_t` pointer to sleep on and a `sleeplock` instead of a `spinlock`. As such, it uses the functions `acquiresleep` and `releasesleep` in its implementation. However, there is a possible deadlock in this implementation. We describe the deadlock with the help of an example.

Suppose two CPUs are concurrently executing the `barriergrouptest` function. Process with pid 3 is executed on CPU C_1 and process with pid 4 is executed on CPU C_2. The processes are executing on two different barrier ids. Both enter the barrier. Both call the `cond_wait` function. Both call `cond_sleep`. **Both acquire their own locks**. Both call `releasesleep`. Both call `wakeup`. Now, 4 iterates through the process table. It **tries to acquire the lock for process** 3 to check its state, but fails since 3 is already acquired by the process itself. Then, 3 iterates through the process table and **tries to acquire the lock for process** 4. This is a deadlock situation. To resolve the same, we make the **acquiring of process lock and the release of the sleeplock** in `condsleep` a critical section using `releasecondsleep_lock`.

8 Regarding Tickslock

There seems to be some deadlock arising due to the `tickslock` lock. More specifically, only the acquiring of `tickslock` in the `wakeup` function seems to be causing issues since commenting out the same prevents any deadlock. We suspect that the issue might be due to the check `if(!holding(&tickslock))` since the comments in `xv6` suggest that this function should only be called while interrupts are off. However all attempts to fix it (for example, by acquiring a dummy lock before calling `holding` and other methods) ultimately failed. Although the problem seems to exclusively stem from the `tickslock` in `wakeup`, we have commented out all instances of it since batch statistics was not necessary in this assignment. The fix in the mail which suggested increasing the `TIMER_INTERVAL` decreased the chances of deadlocks, but still succumbed to larger inputs.