CS330A: Design Assignment 3

Group 15

Aarchie [200004] Kembasaram Nitin [200505]
Harsh Jain [200412] Udit Prasad [201055]

Implementation

Condition variable

- First, we defined condition variable *cond_t* in condvar.h. Its type is empty struct.
- Then we implemented 3 functions to work with conditional variables in condvar.c namely
  1. *cond_wait*(*cond_t* ∗ *cv*, *sleeplock* ∗ *lock*): It simply calls *condlseep(cv,lock)* which put the process to sleep and put the cv into channel of the process.p→chan = cv where cv is conditional variable
  2. *cond_signal*(*cond_t*∗*cv*): It calls *wakeupone(cv)* which in turn wakes up only one process which is sleeping on channel cv.
  3. *cond_broadcast*(*cond_t*∗*cv*): It calls *wakeup(cv)* which in turn wakes up all the process which are sleeping on channel cv

Semaphore

- First we defined semaphore in semaphore.h like this :
  struct semaphore
  {
  int val;
  struct *cond_t* cv;
  struct sleeplock slk;
  }; where val stores the value of semaphore (how many different process can execute the critical section), cv helps in ordering the execution of those processes by using the *slk* sleeplock.
- Then we implemented 3 functions to use semaphores in semaphore.c
  1. *sem_init*(*int v*): It initializes the value of semaphore to v.
  2. *sem_wait*(*semaphore* ∗ *s*): It after acquiring the $s \rightarrow lock$, if value of semaphore is positive,decrements the value of semaphore and releases the lock and thus let the thread enter the critical section otherwise waits for the value to turn positive (waits using *cond_wait*).
  3. *sem_post*(*semaphore* ∗ *s*):First, it acquires the lock of sempahore. Afterwards, It increments the value of semaphore and then calls *cond_signal* which wakes up one process which tries to enter the critical section.

Tests

Before going to the system calls, we defined barrier structure in barrier.h like this:
struct barrier
{
int count;
int allocated;
struct sleeplock *lock*;
struct $cond_t$ cv;
};

where

- count refers to number of processes waiting in that barrier on condition variable.
- allocated is a binary integer which when 0 means the barrier is free and when it is 1, it has been allocated.

- lock is meant for protecting this barrier
- cv is condition variable over which they are waiting

Now moving towards system calls:

1. *barrier_alloc()*: It will go over all the barriers and check if it is allocated or not, if we find one free barrier we will return the index of that and mark it as allocated. In case any barrier is not free, it returns -1.
2. *barrier(instance, id, num)*: Acquires the lock of barrier[id] and increase the count of that barrier by 1. Then it checks if count is equal to num or not, if not then process will wait over barrier[id].cv otherwise it will call *cond_broadcast* which will wakeup all the process waiting in that barrier.
3. $barrier\_free(id)$: It will deallocate barrier[id] by flipping its allocate to 0.

Then for **producer and consumer problem**, we have defined structure of buffer in buffer.h like this:

struct $cond\_buffer\_elem$
{
int x;
int full;
struct *sleeplock* lk;
struct *cond_t* inserted;
struct *cond_t* deleted;
};
where

- x is value stored in buffer
- full is 1 if there is any value stored in buffer otherwise 0
- lk is lock to protect the buffer element while consuming as well as producing
- inserted is condition variable for producing items
- deleted is condition variable for consuming items

Now we will describe the next 3 system calls which were used for item production and consumption.

4. $buffer\_cond\_init()$: In this, we initialize the lock of all the 20 buffer elements. Then we set head and tail to 0 and initialise their respective locks which ensure that head and tail will be changed by any one process only thus protecting them.
5. $cond\_produce(value)$: First acquires the *tail_lock* and then set private index = tail, tail++ and release the *tail_lock* after which it waits for that buffer index to be empty while acquiring the buffer lock of same index, after which it fills that buffer with value. Then it signals by making *cond_signal* call that the buffer at that index is full.
6. *cond_consume*() : It first acquires the headlock and set private index= head,head++ and releases the headlock. Then it acquires the buffer lock of that index and waits for that buffer index to be full, after which it consumes the value by priniting the same using printlock. Then it signals by making *cond_signal* call that the buffer at that index is empty.

Now we have initialised the int $sem\_buffer[10]$ to -1 which we will use for semaphore consumer and producer. Also we will use 4 semaphores for this test namely

- full - used for consumers as it will allow only that many consumers to enter as many indexes are full.
- empty - used for producers as it will allow only that many producers to enter as many indexes are empty.
- con- It is used to increment the *next_producer* index.
- prod- It is used to increment the *next_consumer* index.

Moving to the last 3 system calls:

7. $buffer\_sem\_init()$: It initializes all 4 semaphores with required values : full - 0 (no buffer is full), empty-20(all are empty), con and pro - 1(only one process should increment *next_p* or *next_c*)
8. $sem\_produce(value)$: It checks if there is any empty buffer, if yes then it enters the critical section and acquires the pro lock and then produces the value and increments the *next_p* and calls $sem\_post(pro), sem\_post(full)$.
9. *sem_consume*() : Itchecks if there is any filled buffer , if yes then it enters the critical section, acquires the cons lock and consumes the value and then calls $sem\_post(cons), sem\_post(empty)$.

Comparison

Statistics				
Items per producer	Producers	Consumers	*condprodcons*	*semprodcons*
20	3	2	2	4
20	5	2	4	6
30	4	5	4	7
8	16	8	6	9

From the table, we can see that semaphore one is slower in comparison to purely condition variable implementation. It is because :

- this implementation of producer consumer is not providing concurrency as discussed in the class.
- Number of locks is increased in case of semaphore and we know locks are costly.