CS330 Assignment 3

Jaya Gupta(200471) | Arpit Kumar Rai(200190) | Harshit Bansal(200428) | Avi Kumar(200229)

- Implementation of condsleep and wakeupone
    - **condsleep** - Condsleep implementation is similar to sleep. Two main differences are, first sleeplock is acquired and released instead of spinlock. Second the two function calls to acquire(p->lock) and releasesleep(lk) are kept inside mutual exclusion lock. This is done because two processes are getting into deadlock on calling wakeup before calling their own p->lock.
        - The acquire(p->lock) and releasesleep(lk) should be mutually exclusive, because let say two processes p1 and p2 running in different cpu tries to acquire each other lock in releasesleep -> wakeup() call after acquiring each other lock. This can result in deadlock. So one spinlock is added to guarantee mutual exclusion to this part in condsleep().

    - **wakeupone** - Wakeup is same as wakeup just that we return as soon as we get the first process sleeping on the given condition variable.

      NOTE ON WAKEUP - We observed that myproc(), when called from condsleep() by releasesleep() was returning null address which led to process being trying to acquire its own p->lock, which was leading to deadlock. We dont know the exact reason for this behavior, but we have handled this by adding another condition to check pid of the two processes.

- Condition Variable Implementation

  We implemented Condition Variables in condvar.{c,h} in kernel directory. The implementation involves the following :

  - **Struct cond_t:**
    This struct is defined in condvar.h and contains a single name field which is used to implement the condition variable.

  - **void cond_init(cond_t *cv, char *name)**
    It initializes the condition variable cv with the name passed in name variable.

  - **void cond_wait(cond_t *cv, sleeplock* lock)**
    This function takes two arguments, the first is the condition variable and the second is a lock variable which protects the change of condition (ensures that

condition changes are atomic). This function is called when the condition is false and puts the current process to sleep. Internally it uses the condsleep() function, which first releases the lock and then puts the process to sleep on cv.

- **void cond_signal(cond_t *cv)**
  This function takes a single cond_t * variable as argument and signals one process waiting on cv to enter into RUNNABLE state. Internally it calls the wakeupone() function which traverses the process table and wakes up the first process sleeping on cv. Note that the channel (p->chan) set for sleeping in condition variables is a pointer to the condition variable itself. This function is called when the condition is satisfied and some process waiting for this condition needs to be woken.

- **void cond_broadcast(cond_t *cv)**
  This function is similar to cond_signal with the only difference being that cond_broadcast() wakes up all the processes which are sleeping on cv.

● Barrier System calls

The barrier struct is defined in buffer.h . It has the following structure :

- int state : It determines whether the barrier is free or not.
- Int id : It stores the barrier id.
- Int num_proc_count : It stores the number of processes that have entered the barrier.
- Struct cond_t cv : It is the per barrier condition variable.
- Struct sleeplock cv : It is the lock variable that guards the barrier fields.

The barrier array barrier_list[10] is defined in proc.c
The three system calls are also defined in proc.c

- **barrier_alloc**
  This system call iterates over the barrier_list array and returns the barrier->id of a free barrier. Note that the corresponding barrier locks need to be acquired before checking the state of the current barrier to prevent data-races.

- **barrier**
  This system call implements the barrier using condition variables. It takes three arguments: barrier instance number, barrier array id, and number of processes. It acquires the barrier->lock for the barrier corresponding to passed id, then increments num_proc_count by 1 and then checks whether the condition that all

processes have entered the barrier is satisfied or not. If the condition is not satisfied it calls cond_wait and sends the process to sleep on barrier->cv otherwise it resets num_proc_count to 0 and cond_broadcasts to wakeup all processes sleeping on current barrier->cv. This syscall also prints the “entered barrier … “ and “finished barrier …” messages.

- **barrier_free**
  This system call frees the barrier corresponding to the passed barrier id and resets barrier state and num_proc_count to 0.

Besides the system calls, there is barrierinit() function which is used to initialize the all the barriers in barrier_list at boot time. It initializes state to 0 (free) , num_proc_count to 0 and the condition variable and sleeplock.

Another lock used is print_lock of type sleeplock which is used to prevent jumbled outputs.

● Semaphore Implementation

We implemented Semaphore in semaphore.{c,h} in kernel directory.

- **struct of Semaphore:**
  It contains one condition variable(cv), one integer(x), and one sleeplock(mutex).
- **sem_init:**
  It initializes x with the value passed as an argument, initializes the condition variable(name), and initializes the sleep lock.
- **sem_wait:**
  First, we acquire the sleep lock; if x is less than equal to 0, call cond_wait; if x is positive, decrease the value of x by 1 and release the sleep lock.
- **sem_post:**
  Acquire Sleep lock, increase x by 1, call cond_signal(releases one process), and release the lock.

- Condprodconstest system calls
  Buffer structure is implemented in buffer.h, all system calls and locks used are declared in proc.c
  - **Buffer Structure(buffer_elem[20]):**
    Each buffer_elem contains:
    One integer(x) denotes the value contained, one integer(full) denotes if it is empty or full, one sleeplock(lk), two condition variables (inserted and deleted)
  - Other variables:
    Two global integers(head, tail), and two sleeplocks (grab_head, grab_tail) are declared in proc.c
  - **buffer_cond_init:**

For every element in the buffer array, initializes x=-1, full=0, sleeplock.
Initializes head_grab and tail_grab, head=0, tail=0.

- **cond_produce:**
  Grab a index= tail(keeping grab_tail lock to have mutual exclusion), increases the tail count, Acquire sleeplock of the buffer element, wait till buffer[index] is empty (via deleted condition variable).
  Puts the value in buffer[index].x and make it full, signals inserted condition variable, releases sleeplock(lk)
- **cond_consume:**
  Grab a index= headl(keeping grab_head lock to have mutual exclusion), increases the head count, Acquire sleeplock of the buffer element, wait till buffer[index] is full (via inserted condition variable).
  consume the value in buffer[index].x and make it empty, signals deleted condition variable, releases sleeplock(lk).

- Semprodconstest system calls
  Buffer structure is implemented in sem_buffer.h, all system calls and locks used are declared in proc.c
  - **Buffer Structure(sem_buffer_elem[20]):**
    Each buffer_elem contains:
    One integer(x) denotes the value contained.
  - Other variables:
    Two integers(nextc, nextp), and four semaphores (pro(1), con(1), empty(buffer_count),full(0)).
    pro and con semaphore are used for mutual exclusion which incrementing nextp, nextc respectively.
  - **buffer_sem_init:**
    For every element in the buffer array, initializes x=-1.
    Initializes nextc=0, nextp=0.
    Initializes the four semaphores with values written above.
  - **sem_produce:**
    Waits for empty semaphore(checking if there are empty space in buffer); grab a index= nextp(using pro semaphore), increases the nextp count, post on full semaphore.
  - **cond_consume:**
    Waits for full semaphore(checking if there is some not consumed value in buffer); grab a index= nextc(using con semaphore), increases the nextc count, post on empty semaphore.

- Time comparison between Condprodconstest and Semprodconstest

  The following data is taken without including print statements.

Input			Time for condprodconstest	Time for semdprodconstest
#item/producer	**#producer**	**#consumer**		
20	3	2	1	1
40	10	10	1	4
500	3	2	6	11
500	20	20	45	80

It is quite clear from the above comparisons that conditional variables are much faster than semaphore implementation.
The reasons behind that are-

- Condition variables provide better concurrency as there can be multiple producer-consumer read-write in condition variable implementation. While semaphore implementation provides only one read-write at once (as production/consumption of a element is in a lock for mutual exclusion).