Assignment 3

Abhinav Kumar Singh
Harshit Gupta
Ronit Mittal

1 Implementation of condition variable

The structure of condition variable is:

```
struct cond_t{
    uint64 i;               // i=1; condition has been been fulfilled
    struct sleeplock lk;
};
```

1.1 cond wait

```
void cond_wait (struct cond_t *cv, struct sleeplock *lock){
    releasesleep(lock);
    acquiresleep(&cv->lk);
    condsleep(cv, &cv->lk);
    releasesleep(&cv->lk);
    acquiresleep(lock);
    return;
}
```

The function expects an acquired sleeplock. Since the thread running(which called this function) is currently unable to proceed, we release the lock and acquire the condition variable's sleeplock and call condsleep. The thread sleeps in the condition variable's channel and will wake up only when a thread on this channel calls wakeup or wakeupone.

1.2 cond signal

```
void cond_signal (struct cond_t *cv){
    acquiresleep(&cv->lk);
    cv->i = 1;
    releasesleep(&cv->lk);
    wakeupone(cv);
    acquiresleep(&cv->lk);
    cv->i = 0;
    releasesleep(&cv->lk);
    return;
}
```

After the thread executes and the stream is available for another thread, the process calls this function, which first updates the condition variable's value to 1 indicating that the stream is available for another thread, then it calls wakeupone which wakes up one thread on the same channel and after successfully waking it up, it again updates the value to 0, indicating that a thread is already running.

1.3 cond broadcast

```
void cond_broadcast (struct cond_t *cv){
    acquiresleep(&cv->lk);
    cv->i = 1;
    releasesleep(&cv->lk);
    wakeup(cv);
    acquiresleep(&cv->lk);
    cv->i = 0;
    releasesleep(&cv->lk);
}
```

The function is similar to cond signal. Instead of waking up a single process, it wake sup all the processes on the channel. It is required in some cases to use broadcast instead of simple signal.

2 Implementation of semaphore

The structure of semaphore is:

```
struct semaphore{
    int val;             // sem_wait() decrements, sem_post() increments the value
    struct sleeplock lk;    // lock for updating values
    struct cond_t c;
};
```

2.1 sem init

```
void sem_init (struct semaphore *s, int x){
    initsleeplock(&s->lk,"lock");
    initsleeplock(&s->c.lk,"lock");
    s->val=x;
    return;
}
```

Initializes the value of all the attributes of the structure.

2.2 sem wait

```
void sem_wait (struct semaphore *s){
    acquiresleep(&s->lk);
```

```
    s->val=s->val-1;
    while(s->val<0) cond_wait(&s->c,&s->lk);
    releasesleep(&s->lk);
    return;
}
```

The wait() function decreases the value of semaphore on each call and then checks if the thread can currently run or not. Value less than zero sends the thread to sleep. We have used while loop to avoid execution for cases where a signal is received by the thread but the value of the semaphore is negative.

2.3 sem post

```
void sem_post (struct semaphore *s){
    acquiresleep(&s->lk);
    s->val=s->val+1;
    cond_signal(&s->c);
    releasesleep(&s->lk);
}
```

Increases the value of semaphore after the current thread has successfully executed and signals a sleeping thread on the channel to wake up. Of course, only if the value of the semaphore is greater than or equal to zero, it is able to run. Else the thread again goes to sleep.

3 System Calls

3.1 cond sleep

This function is same as sleep system call. It takes a condition variable and a sleeplock as input instead of void* and spinlock.

3.2 wakeupone

Similar to wakeup but returns as soon as a process wakes up on the channel.

3.3 barrier alloc

```
int barrier_alloc(void){
  for (int i = 0; i < 10; i++){
    acquiresleep(&s_lock);
    if (barriers[i] == 0){
      barriers[i] = 1;
      releasesleep(&s_lock);
      return i;
    }
    releasesleep(&s_lock);
```

```
  }
  return -1;
}
```

Iterates through all the elements of the abrries and tries to fin a free barrier. A value of zero indicates the barrier for that index is free and occupied if one. We use a sleeplock here to avoid race condition. And finally we return the index of the free barrier if found. Returns -1 if no barrier is free.

3.4 barrier

```
void barrier(int barrier_inst_num, int barrier_arr_id, int num_proc){
  struct proc *p = myproc();
  int pid;
  acquire(&p->lock);
  pid = p->pid;
  release(&p->lock);
  acquiresleep(&s_lock);
        \\ print stuff
  releasesleep(&s_lock);
  return;
}
```

We sue p lock to extract the process's information and acquire a sleep lock so that the output is not jumbled.

3.5 barrier free

```
void barrier_free(int id){
  acquiresleep(&s_lock);
  barriers[id] = 0;
  releasesleep(&s_lock);
  return;
}
```

Frees the barrier with input id. Sleeplock required to avoid race condition.

3.6 buffer cond init

```
void buffer_cond_init(){
  initsleeplock(&lock_delete, "lock_delete");
  initsleeplock(&lock_insert, "lock_insert");
  initsleeplock(&lock_print, "lock_print");
  for (int i = 0; i < SIZE; i++){
    initsleeplock(&(buffer[i].lock), "buffer_lock");
    buffer[i].data = 0;
    buffer[i].full = 0;
    buffer[i].inserted.i = 0;
```

```
    initsleeplock(&buffer[i].inserted.lk, "lock");
    buffer[i].deleted.i = 0;
    initsleeplock(&buffer[i].deleted.lk, "lock");
  }
  head = 0;
  tail = 0;
  return;
}
```

Initializes all the locks and sets all the barrier attributes to zero.

3.7 cond produce

```
void cond_produce(int p){
  acquiresleep(&lock_insert);
  int index = tail;
  tail = (tail + 1) % SIZE;
  releasesleep(&lock_insert);
  acquiresleep(&buffer[index].lock);
  while (buffer[index].full == 1){
    cond_wait(&buffer[index].deleted, &buffer[index].lock);
  };
  buffer[index].data = p;
  buffer[index].full = 1;
  cond_signal(&buffer[index].inserted);
  releasesleep(&buffer[index].lock);
  return;
}
```

We use the lock insert sleeplock to update tail of buffer. Each buffer element has its own sleeplock which is acquired before accessing it's elements. If the buffer is full, we simply sleep the thread by calling cond wait. Else/on receiving a signal, we insert new data to the index and signal for the consumer thread to wake up (the thread which has been waiting for the value to be inserted).

3.8 cond consume

```
int cond_consume(){
  int v, index;
  acquiresleep(&lock_delete);
  index = head;
  head = (head + 1) % SIZE;
  releasesleep(&lock_delete);
  acquiresleep(&buffer[index].lock);
  while (buffer[index].full == 0){
    cond_wait(&buffer[index].inserted, &buffer[index].lock);=
  };
```

```
  v = buffer[index].data;
  buffer[index].full = 0;
  cond_signal(&buffer[index].deleted);
  releasesleep(&buffer[index].lock);
  acquiresleep(&lock_print);
  printf("%d ", v);
  releasesleep(&lock_print);
  return v;
}
```

This system call is for consumers. Here we first acquired sleeplock so that we can avoid race condition and thenincreased the head variable. Then we acquired the buffer array index lock so that no other consumer try to write on the same array inducing race condition. After that there is a condition to check if buffer is empty. If yes then consumer is put to wait until further signal. After waking, buffer index is updated and a signal is made to announce the buffer is empty again. After that it prints the value followed by releasing the locks.

3.9 buffer sem init

```
void buffer_sem_init(){
  initsleeplock(&bb_lock_delete, "lock_delete");
  initsleeplock(&bb_lock_insert, "lock_insert");
  initsleeplock(&bb_lock_print, "lock_print");
  for (int i = 0; i < SIZE; i++)
  {
    initsleeplock(&(bounded_buffer[i].lk), "buffer_lock");
    bounded_buffer[i].data = 0;
    bounded_buffer[i].full = 0;
    sem_init(&(bounded_buffer[i].inserted), 0);
    sem_init(&(bounded_buffer[i].deleted), 1);
  }
  bb_head = 0;
  bb_tail = 0;
  return;
}
```

This system call is used to initialize buffer. We initiated all the required locks as well as the buffer. We also initiated all the required variables.

3.10 sem produce

```
void sem_produce(int p){
  acquiresleep(&bb_lock_insert);
  int index = bb_tail;
```

```
  bb_tail = (bb_tail + 1) % SIZE;
  releasesleep(&bb_lock_insert);
  sem_wait(&bounded_buffer[index].deleted);
  acquiresleep(&bounded_buffer[index].lk);
  bounded_buffer[index].data = p;
  bounded_buffer[index].full = 1;
  releasesleep(&bounded_buffer[index].lk);
  sem_post(&bounded_buffer[index].inserted);
  return;
}
```

This system call is for producer in semaphore implementation. First we acquired the insert lock and then increased the value of tail. After that we have to wait till there is space in buffer index. After all this we also send the signal using sem post method.

3.11 sem consume

```
int sem_consume(){
  acquiresleep(&bb_lock_delete);
  int index = bb_head;
  bb_head = (bb_head + 1) % SIZE;
  releasesleep(&bb_lock_delete);
  sem_wait(&bounded_buffer[index].inserted);
  acquiresleep(&bounded_buffer[index].lk);
  int v = bounded_buffer[index].data;
  releasesleep(&bounded_buffer[index].lk);
  sem_post(&bounded_buffer[index].deleted);
  acquiresleep(&bb_lock_print);
  printf("%d ", v);
  releasesleep(&bb_lock_print);
  return v;
}
```

This system call is for consumers in semaphore implementation. Here we first acquired sleeplock so that we can avoid race condition and then increased the head variable. Then we used sem wait function to check for the signal. Then we acquired the buffer array index lock so that no other consumer try to write on the same array inducing race condition. After that we send the singal using sem post to announce that the buffer array index is now empty. After that we print the value of consumed item.

4 Observation

These are the observations for a few runs:

```
exec arg1 arg2 arg3 time
condprodconstest 1 1 1 1
semprodconstest 1 1 1 1
```

```
condprodconstest 5 1 1 2
semprodconstest 5 1 1 2
condprodconstest 10 2 2 4
semprodconstest 10 2 2 5
condprodconstest 20 4 2 13
semprodconstest 20 4 2 15
```

We observed that semprodconstest took slightly more time than condprodconstest for same inputs. And the difference increases for increasing number of threads involved in the process. This is probably due to the fact that implementation of semaphore already includes condition variable and is a complex structure than much simpler condition variable. So the semaphore runs through more lines of codes than condition variable in the system calls.