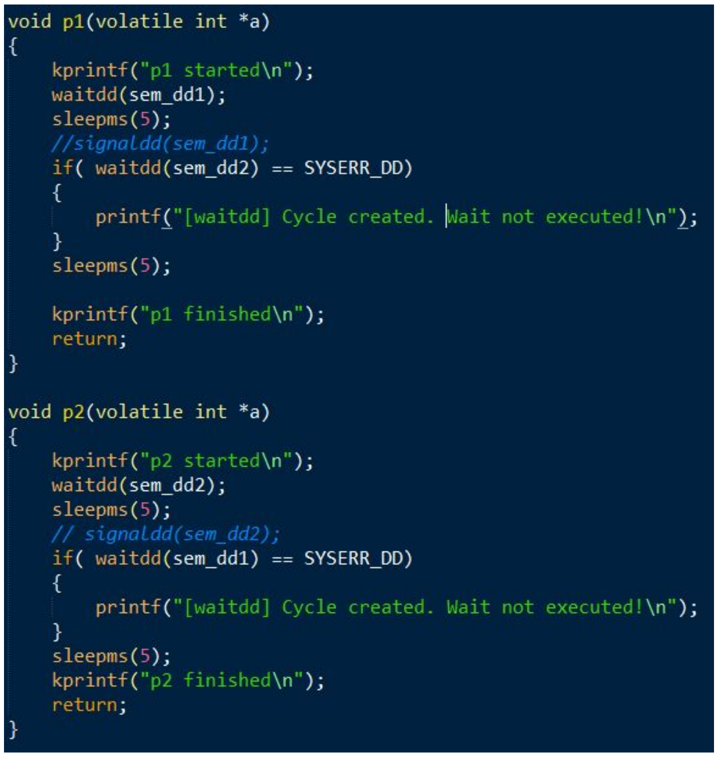
Krutarth Rao
(0027262283)

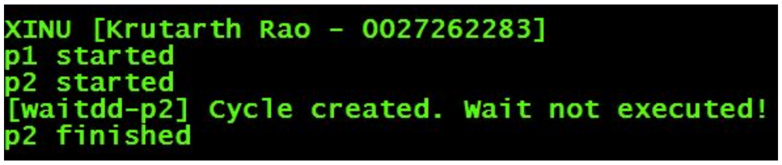
3

Deadlock detection is implemented using cycle detection in a resource graph. The data structure is defined in *deadlock.h* and the operations on the data structure are defined in *deadlock.c*. The resource graph is a directed graph where an edge from a semaphore to a process means the semaphore is held by the process and an edge from a process to a semaphore is interpreted as a process waiting on the semaphore.
In *waitdd*, if a process is put into the waiting state, an edge is added to the resource graph from the blocked process to the semaphore. Otherwise, an edge is added from the semaphore to the calling process. In both cases, if the addition of the edge to the resource graph creates a cycle, the edge is removed and *SYSERR_DD* is returned.
In *signaldd*, the edge from the semaphore to the calling process is removed since the process no longer holds the semaphore. Also, If a process needs to be released as a result of the *signaldd*, the edge from the process to the semaphore is removed since the process is no longer waiting on the semaphore.

Test case 1 (2 process & 2 (1-count) semaphores)



```
void p1(volatile int *a)
{
    kprintf("p1 started\n");
    waitdd(sem_dd1);
    sleepms(5);
    //signaldd(sem_dd1);
    if( waitdd(sem_dd2) == SYSERR_DD)
    {
        printf("[waitdd] Cycle created. Wait not executed!\n");
    }
    sleepms(5);

    kprintf("p1 finished\n");
    return;
}

void p2(volatile int *a)
{
    kprintf("p2 started\n");
    waitdd(sem_dd2);
    sleepms(5);
    // signaldd(sem_dd2);
    if( waitdd(sem_dd1) == SYSERR_DD)
    {
        printf("[waitdd] Cycle created. Wait not executed!\n");
    }
    sleepms(5);
    kprintf("p2 finished\n");
    return;
}
```



```
XINU [Krutarth Rao - 0027262283]
p1 started
p2 started
[waitdd-p2] Cycle created. Wait not executed!
p2 finished
```

Output:
When the processes are resumed in order of p1 -> p2, the last wait is not executed since waitdd detects the cycle and returns the *SYSERR_DD* instead of letting the process deadlock. Once the *signaldd* is uncommented, there are no deadlocks detected.

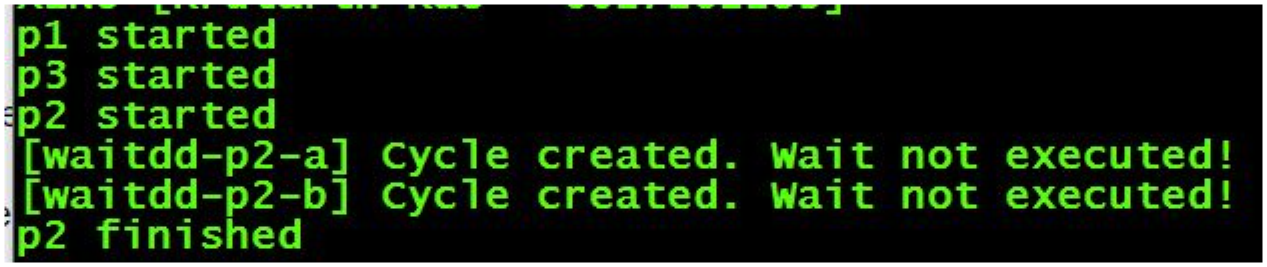
Test Case 2 (3 processes and 3 (1-count) semaphores)

Using three processes with the following code:

```
void p1(volatile int *a)
{
      kprintf("p1 started\n");
      waitdd(sem_dd1);
      sleepms(5);
      //signaldd(sem_dd1);
      if( waitdd(sem_dd2) == SYSERR_DD)
            printf("[waitdd-p1-a] Cycle created. Wait not executed!\n");
      sleepms(5);
      //signaldd(sem_dd2);
      if( waitdd(sem_dd3) == SYSERR_DD)
            printf("[waitdd-p1-b] Cycle created. Wait not executed!\n");

      kprintf("p1 finished\n");
      return;
}

void p2(volatile int *a)
{
      kprintf("p2 started\n");
      waitdd(sem_dd2);
      sleepms(5);
      // signaldd(sem_dd2);
      if( waitdd(sem_dd1) == SYSERR_DD)
            printf("[waitdd-p2-a] Cycle created. Wait not executed!\n");
      sleepms(5);
      // signaldd(sem_dd1);
      if( waitdd(sem_dd3) == SYSERR_DD)
            printf("[waitdd-p2-b] Cycle created. Wait not executed!\n");

      kprintf("p2 finished\n");
      return;
}

void p3(volatile int *a)
{
      kprintf("p3 started\n");
      waitdd(sem_dd3);
      sleepms(5);
      // signaldd(sem_dd3);
      if( waitdd(sem_dd1) == SYSERR_DD)
            printf("[waitdd-p3-a] Cycle created. Wait not executed!\n");
      sleepms(5);
```

```
        // signaldd(sem_dd1);
        if( waitdd(sem_dd2) == SYSERR_DD)
              printf("[waitdd-p3-b] Cycle created. Wait not executed!\n");

        kprintf("p3 finished\n");
        return;
}
```

The processes are resumed in order p1, p3 and p2. The output is:

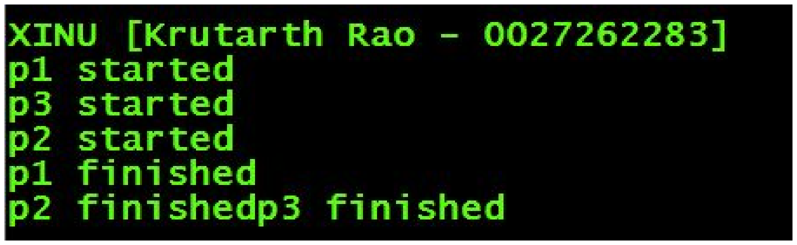


```
p1 started
p3 started
p2 started
[waitdd-p2-a] Cycle created. Wait not executed!
[waitdd-p2-b] Cycle created. Wait not executed!
p2 finished
```

The deadlock is detected in the second and third waitdd call by process 2. As a result, the wait is not executed and process 2 reaches completion without getting blocked. Also, since p2 exits without releasing the semaphores, the remaining processes are left in a blocking state and are successfully ignored by the deadlock detection functionality.

Test Case 3 (Own)

Although testing will only be done with 1-count semaphores according to TA notes, it is important to test if the detection functionality avoids a false positive when the count is greater than 1. For the test above, there should be no deadlock once the semaphores being used are created with a count of 3.



```
XINU [Krutarth Rao - 0027262283]
p1 started
p3 started
p2 started
p1 finished
p2 finishedp3 finished
```

The output above shows that once the semaphores being used are created with a count of 3, no deadlock is detected and the detection continues to function correctly for sem counts greater than 1.

Less drastic actions compared to closing application

Instead of closing the application, the app programmer may switch to the process/method currently holding the needed semaphore and ask the holding process to signal the semaphore. Now the original process may attempt to acquire the semaphore again. If the programmer doesn't know the method/process holding the semaphore, it may choose to use skip the critical section if the program's functionality allows.

4

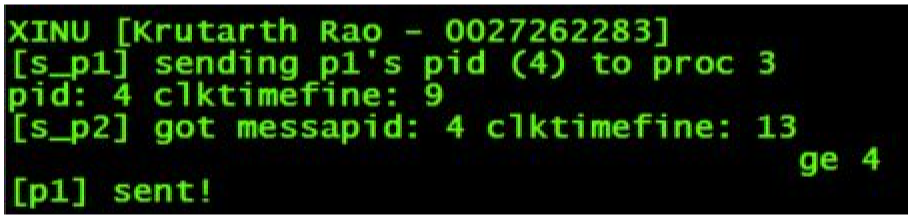
To maintain a queue of processes waiting to send a message to process, an entry *struct array_queue sw_queue* was added to the process table entry. The field is initialized as an empty queue when the process is created. A second field *char sndflag* is used to let the sending process know if the recipient has read the message. The flag is set to 1 when a process has to go into *SENDWAIT* state and once the receiver dequeues the process and reads the message, the flag is set to 0. During this time, if the sending process is woken up before the recipient gets a chance to see the message, the sending process will return from *reschd* with the flag still at 1. If the the flag is not 0, the sender can assume the message was not read in time and can return *TIMEOUT*. If maxwait was negative, the *sendbk* method goes into an loop of *reschd* calls till the message is read.
The changes needed to accomplish & test the above:

- 3 new fields in process table entry
- Modification of *receive*() to handle processes that may be waiting to send to current process.
- New *sendbk* method.
- Modified *create()* to initialize the FIFO queue used to hold the waiting processes.
- Redid Lab3 section to bring back millisecond time keeping
- Created test cases in testSendBlocking.c

The runtime overhead of the new queue data structure is constant when enqueuing and dequeuing due to the wrap around head and tail indexes *(see array_queue.c)*. The memory overhead is to keep a preallocated array to store pid's of processes that may needed to be added in the future to the waiting queue.

Test case 1 (simple functionality)

Two processes are created and sendbk is used to send a message(it's own pid) to a process with an empty buffer. Test can be found in test1() in testSendBlocking.c. Output:

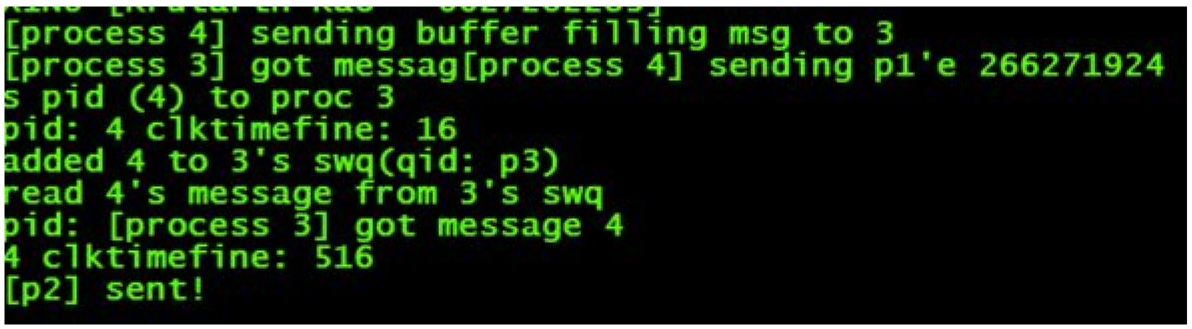


```
XINU [Krutarth Rao - 0027262283]
[s_p1] sending p1's pid (4) to proc 3
pid: 4 clktimefine: 9
[s_p2] got messapid: 4 clktimefine: 13
                                    ge 4
[p1] sent!
```

The message is sent successfully and printed by the recipient (text scattered due to context switching).

Test 2 (recipient with full buffer without timeout)

Two processes are created and the recipient buffer is filled with a dummy message by sender. The second sendbk is made such the sendbk doesn't return TIMEOUT but the recipient sleeps for 500ms before calling the second recieve, the sending process should block till then. This test checks if sendbk will block when the buffer is full. Outptut:

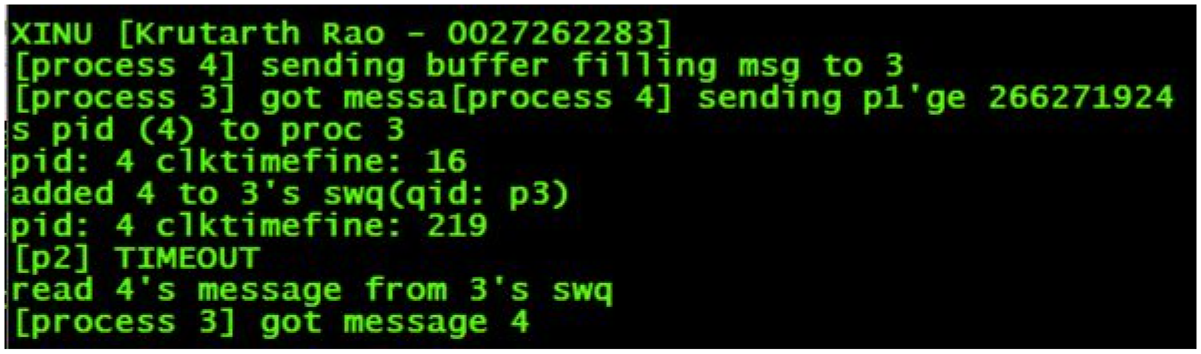


```
[process 4] sending buffer filling msg to 3
[process 3] got messag[process 4] sending p1'e 266271924
s pid (4) to proc 3
pid: 4 clktimefine: 16
added 4 to 3's swq(qid: p3)
read 4's message from 3's swq
pid: [process 3] got message 4
4 clktimefine: 516
[p2] sent!
```

The time elapsed after sendbk returns is approx. the time the recipient process slept for before calling the second receive.

Test 3 (recipient with full buffer and timeout)

Two processes are created and the recipient buffer is filled with a dummy message by sender. The call to sendbk is made such the sendbk returns TIMEOUT. This test checks if sendbk will block and timeout when the buffer is full. This is similar to the test above but recipients sleeps for longer than sender can wait Outptut:



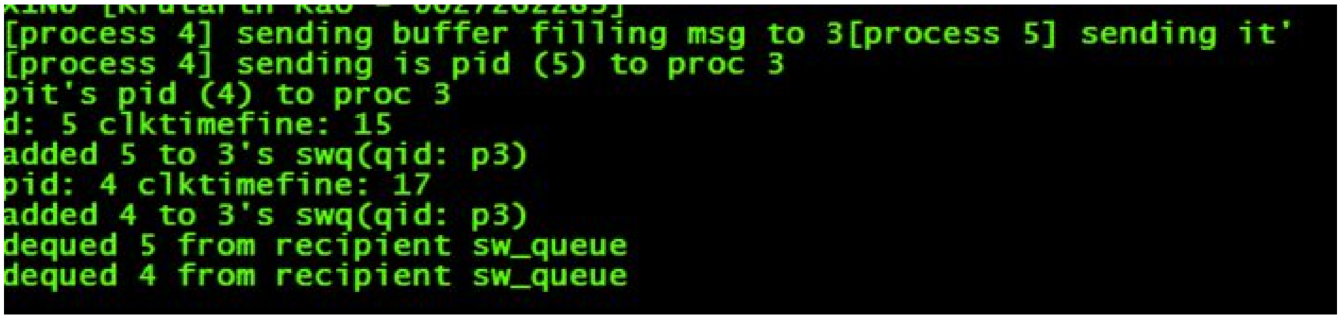
```
XINU [Krutarth Rao - 0027262283]
[process 4] sending buffer filling msg to 3
[process 3] got messa[process 4] sending p1'ge 266271924
s pid (4) to proc 3
pid: 4 clktimefine: 16
added 4 to 3's swq(qid: p3)
pid: 4 clktimefine: 219
[p2] TIMEOUT
read 4's message from 3's swq
[process 3] got message 4
```

The sending process times out as seen by the clktimefine after the return from sendbk.

NOTE: The handout doesn't specify whether the recipient should still read a message delivered by a timed out sender. This implementation of sendbk & receive will have the recipient read the pending messages regardless of the sender being timedout. This can be changed by adding a new field in the process table entry *char proc_timedout* that is set to 1 by the sender before timing out and the recipient will check this flag before reading the message.

Test 4 (infinite blocking)

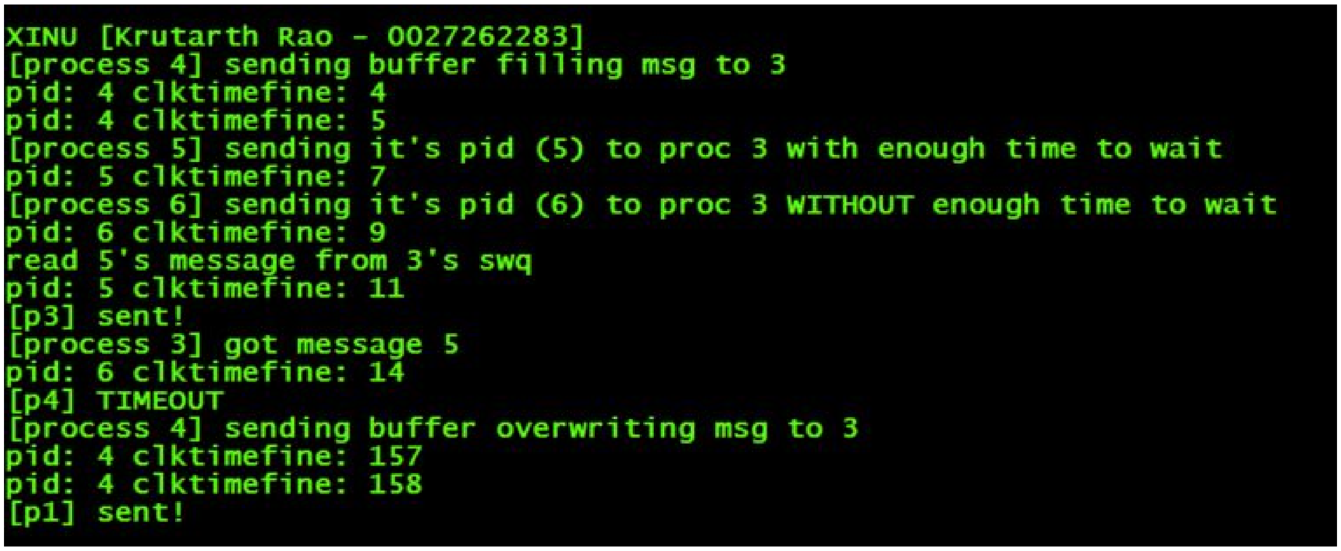
Three processes are created. Two of these processes attempt to send a message while the recipient gets stuck in an infinite loop before calling receive. After a few seconds, main prints the sw_queue of the recipient process to find the senders in it and the senders need to be killed since they don't return (as seen by the absence of the *clktimefine* printed after return) . Output:



```
XINU [Krutarth Rao - 0027262283]
[process 4] sending buffer filling msg to 3[process 5] sending it'
[process 4] sending is pid (5) to proc 3
pit's pid (4) to proc 3
d: 5 clktimefine: 15
added 5 to 3's swq(qid: p3)
pid: 4 clktimefine: 17
added 4 to 3's swq(qid: p3)
dequed 5 from recipient sw_queue
dequed 4 from recipient sw_queue
```

Test 5 (mix of valid, timed out and overwriting senders with random timeout)

(see test5() in testSendBlocking.c) 4 processes are created. 3 are senders. Each sender exhibits a seperate behaviour of sending. Namely sending by waiting, overwriting/filling and timing out. This test confirms that multiple processes may use *sendbk* in their own ways but the recipient obtains the intended messages. This test case most closely resembles a Output:



```
XINU [Krutarth Rao - 0027262283]
[process 4] sending buffer filling msg to 3
pid: 4 clktimefine: 4
pid: 4 clktimefine: 5
[process 5] sending it's pid (5) to proc 3 with enough time to wait
pid: 5 clktimefine: 7
[process 6] sending it's pid (6) to proc 3 WITHOUT enough time to wait
pid: 6 clktimefine: 9
read 5's message from 3's swq
pid: 5 clktimefine: 11
[p3] sent!
[process 3] got message 5
pid: 6 clktimefine: 14
[p4] TIMEOUT
[process 4] sending buffer overwriting msg to 3
pid: 4 clktimefine: 157
pid: 4 clktimefine: 158
[p1] sent!
```

NOTE: The output above was generated by disabling interrupts in *kprintf*.

The output shows that all senders except the sender with a timeout of (maxwait = 1) successfully send their messages either through the queue or through the buffer directly.
Also, the sender intended to time out occasionally sends the message without timing out. This is because it is scheduled right after the buffer is cleared and no other process has written anything to

the buffer so it doesn't have to wait. In this case, the recipient successfully reads the delivered message.

Bonus

The *is_cycle* method in deadlock.c currently returns a boolean result for a cycle in the graph. The algorithm can be changed to use strongly connected component detection to determine the edges in the cycle. Once the *is_cycle* method returns the edges responsible for the cycle to *waitdd*, *waitdd* will then be changed to return a struct:

```
Struct wait_return    {
     int   status;    /* DEADLOCK or WAIT_OK     */
     int dependencies[]  /* The edges of the graph causing a cycle*/
};
```

The app programmer may call *waitdd* to get the above struct and if the status of the returned struct is DEADLOCK, the programmer can iterate through the dependencies array of the struct to obtain the semid and pid of the semaphores and processes involved in the cycle. On the other hand, only the pid's of the processes causing the deadlock may be returned since the *prsem* field in the process table entry holds the semaphore each process is waiting on and the app programmer may fetch the values from the process table themselves.

If the exact cycle needs to be determined, a graph data structure as defined in deadlock.h may be returned by *waitdd* instead of the *dependencies* array. It is then up to the app programmer to interpret the graph. The vertices of the graph will continue to be the semaphores and pid of the processes as used in the resource graph. The returned graph will only be the subgraph containing the cycle.