Semaphore with Blocking

```
class Semaphore {
   int value;
   ProcessList pl;

   void down () {
      value -= 1;
      if (value < 0) {
         // add this process to pl
         pl.enqueue(currentProcess);
         Sleep();
      }
   }

   void up () {
      Process P;
      value += 1;
      if (value <= 0) {
         // remove a process P from pl
         P = pl.dequeue();
         Wakeup(P);
      }
   }
}
```

Producer/Consumer with Semaphores

```
const int n;
Semaphore empty(n),full(0),mutex(1);
Item buffer[n];
```

Producer

```
int in = 0;
Item pitem;
while (1) {
  // produce an item
  // into pitem
  empty.down();
  mutex.down();
  buffer[in] = pitem;
  in = (in+1) % n;
  mutex.up();
  full.up();
}
```

Consumer

```
int out = 0;
Item citem;
while (1) {
  full.down();
  mutex.down();
  citem = buffer[out];
  out = (out+1) % n;
  mutex.up();
  empty.up();
  // consume item from
  // citem
}
```

Producer/Consumer

```
#include <semaphore.h>

#define N 10
int buffer[N];
int counter = 0, in = 0, out = 0, total = 0;

sem_t semmutex; // sem_init(&semmutex, 0, 1); in main()
sem_t semfull;  // sem_init(&semfull, 0, 0); in main()
sem_t semempty; // sem_init(&semempty, 0, N); in main()
```

```
void *producer(void *junk) {
    while(1) {
        sem_wait(&semempty);
        sem_wait(&semmutex);

        buffer[in] = total++;
        printf("Produced: %d\n",
                            buffer[in]);
        in = (in + 1) % N;
        counter++;

        sem_post(&semmutex);
        sem_post(&semfull);
    }
}
```

```
void *consumer(void *junk) {
    while(1) {
        sem_wait(&semfull);
        sem_wait(&semmutex);

        printf("Consumed: %d\n",
                                buffer[out]);
        out = (out + 1) % N;
        counter--;

        sem_post(&semmutex);
        sem_post(&semempty);
    }
}
```

Deadlock!

```
#include <semaphore.h>

#define N 10
int buffer[N];
int counter = 0, in = 0, out = 0, total = 0;

sem_t semmutex; // sem_init(&semmutex, 0, 1); in main()
sem_t semfull;  // sem_init(&semfull, 0, 0); in main()
sem_t semempty; // sem_init(&semempty, 0, N); in main()
```

```
void *producer(void *junk) {
    while(1) {
        sem_wait(&semmutex);
        sem_wait(&semempty);

        buffer[in] = total++;
        printf("Produced: %d\n",
                            buffer[in]);
        in = (in + 1) % N;
        counter++;

        sem_post(&semfull);
        sem_post(&semmutex);
    }
}
```

```
void *consumer(void *junk) {
    while(1) {
        sem_wait(&semmutex);
        sem_wait(&semfull);

        printf("Consumed: %d\n",
                                buffer[out]);
        out = (out + 1) % N;
        counter--;

        sem_post(&semempty);
        sem_post(&semmutex);
    }
}
```

Counting Semaphore

Mutex

A simplified version of a Semaphore that can only be locked or unlocked

Mutex

Shared variables

```
Semaphore mutex;
```

Code for process P_i

```
while (1) {
  down(mutex);
  // critical section
  up(mutex);
  // remainder of code
}
```

Monitors

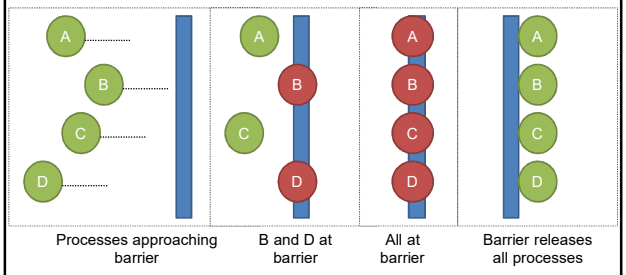
```
class ProducerConsumer {
  private static final int n;
  Item buffer[] = new Item[n];
```

```
public synchronized Item consumer() {
  while (count == 0) {
    try {
      wait();
    }
    catch (InterruptedException e) {
      System.err.println("interrupted");
    }
  }
  cItm = buffer[out];
  out = (out + 1) % n;
  count-=1;
  if (count == n-1) {
    // wake up the producer
    notify();
  }
  return cItm;
}
```

```
public synchronized void producer() {
  //produce an item into pItm
  while (count == n) {
    try {
      wait();
    } catch (InterruptedException e) {
      System.err.println("interrupted");
    }
  }
  buffer[in] = pItm;
  in = (in + 1) % n;
  count+=1;
  if (count == 1) {
    // wake up the consumer
    notify();
  }
}
```

Message Passing

Barriers



Dining Philosophers