# Preview

- Producer-Consumer Problem
  - Race condition in Producer-Consumer Problem
- Semaphores
  - Concept of Semaphores
  - Semaphore Operation
  - Producer-Consumer Problem with Semaphore
- Dinning Philosopher's Problems



# The Producer-Consumer Problem

- The producer-consumer problem is a classic example of a multi-process synchronization problem

**Description**

- Two processes (or threads) share a common, fixed-sized buffer.
- Producer puts information into the buffer, and consumer takes it out.

**Troubles arises**

- When the producer wants to put a new item in the buffer, but it is already full.
- When the consumer tries to take a item from the buffer, but buffer is already empty.



# The Producer-Consumer Problem

- When the producer wants to put a new item in the buffer, but it is already full.
  - Solution – producer is go to sleep, awakened by consumer when consumer has removed on or more items.
- When the consumer tries to take a item from the buffer, but buffer is already empty.
  - Solution – consumer is go to sleep, awakened by the producer when producer puts one or more information into the buffer.



# The Producer-Consumer Problem

```
#define N 100
int count = 0;
void producer()
{
   int item
   while (true)
   {
       item = produce_item();
       if (count == N)
             sleep();
       insert_item(item)
       count = count + 1;
       if (count ==1)
            wakeup(consumer);
   }
}
```

```
void consumer()
{
   int item;
   while(true)
   {
        if (count == 0)
              sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1)
             wakeup(producer);
        consume_item(item);
   }
}
```



# Race condition
## (in producer-consumer problems)

1. At time $T_0$ buffer is empty (count = 0)
2. The consumer just read count = 0, since the consumer's CPU time is over, scheduler assign a CPU time to producer.
3. Producer produce item and check count, count = 0. insert item to buffer. Increase count = count +1. since count =1, it calls wakeup(consumer). Since the consumer is not sleeping yet, consumer miss the wakeup signal.
4. The consumer get CPU time. Consumer already read count =0, consumer go to sleep
5. the producer keep produce items and finally buffer become full. The producer go to sleep



# Semaphores – by E. W. Dijkstra

- A semaphore is an integer variable which could have value
  - 0:  no wakeups are saved
  - + i: i wakeups are pending
- A semaphore is accessed only through two standard atomic operations *down* (or P) and *up* (or V).

## Concept of Semaphores

- Modification to the integer value of the semaphore in the down and up operations are executed indivisibly.
- Which means that when a process is modifying the semaphore value, no other process can simultaneously modify that same semaphore value.



## Semaphore Operation

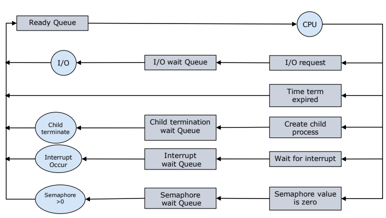
```
void down (S)
{
   If S ≤ 0
   {
     1. Add this process to the
        sleeping list (queue)
     2. block;
   }
   S = S – 1;
}
```

```
void up (S)
{
   S = S + 1;
   If S = 1
   {
     1. choose one process P
        from the sleeping list
     2. wakeup(P) to finish down
        operation
   }
}
```



## Process Scheduling

(Scheduling Queues)




Queueing-diagram of process scheduling



## Semaphore Implementation

The normal way for implementing a semaphore

- Implement semaphore operations ***up*** and ***down*** as system call.
- operating system briefly disabling all interrupts while it is testing the semaphore, updating it and putting the process to sleep.



## Usages of semaphore

```
semaphore mutex = 1
repeat
   down (mutex);
      Critical Section
   up (mutex);
      Remainder Section
until false
```

```
void down (S)
{
   If S ≤ 0
   {
   1. Add this process to the sleeping list
   2. block;
   }
   S = S – 1;
}

void up (S)
{
   S = S + 1;
   If S = 1
   {
   1. choose one process P from the
      sleeping list
   2. wakeup(P) to finish down operation
   }
}
```



## Solving the Producer-Consumer Problem using Semaphores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
void producer ()
{
   int item;

   while (ture)
   {
      item = produce_item();
      down (&empty);
      down (&mutex);
      insert_item(item);
      up(&mutex);
      up(&full);
   }
}
```

```
void consumer()
{
   int item;

   while (true)
   {
      down(&full)
      down(&mutex)
      item = remove_item();
      up(&mutex);
      up(&empty);
      consume_item(item);
   }
}
```

# Dining Philosophers Problem