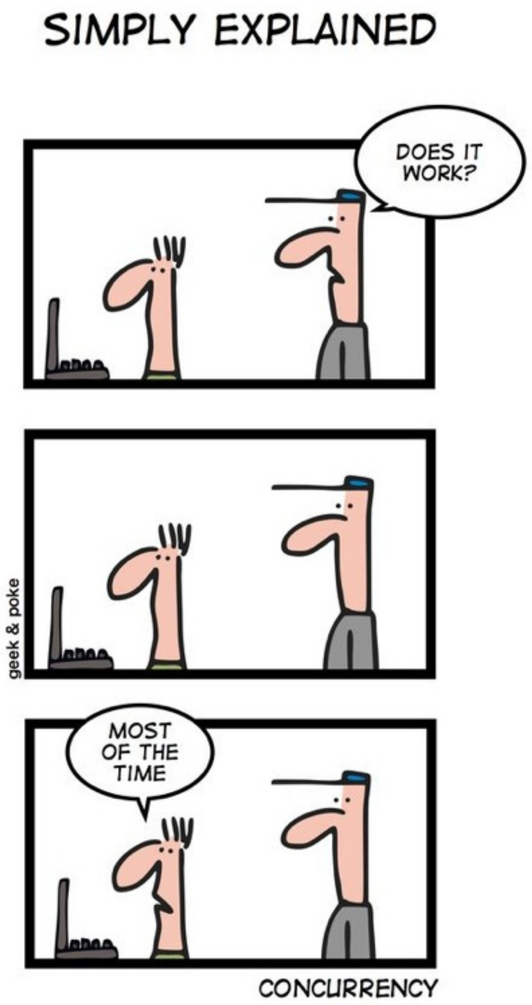
Concurrency (Part III):
Mutual Exclusion, **Synchronization**, Deadlock, and Starvation

Professor Travis Peters
CSCI 460 Operating Systems
Fall 2019

Some slides & figures adapted from Stallings instructor resources.

Some slides adapted from Adam Bates's F'18 CS423 course @ UIUC
https://courses.engr.illinois.edu/cs423/sp2018/schedule.html



–http://www.datamation.com/news/tech-comics-quantum-physics-2.html

Semaphores

Semaphores = signalling variables

A process/thread stops at a certain point until it is signalled to proceed.

A variable that has an integer value upon which only three operations are defined:

- A semaphore may be initialized to a **nonnegative integer value**
- The `semWait` (or `down` or `P=test`) operation…
  - **decrements** the semaphore value
  - if **value is less than 0 — block** and **wait for a signal**; else **continue**
- The `semSignal` (or `up` or `V=increment`) operation…
  - **increments** the semaphore value
  - if **value is less than or equal to 0**, ***transmit a signal*** to unblock a waiting thread

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

See Also: Mutexes vs. Binary Semaphore (vs. General /Counting Semaphores)

Semaphores

Semaphores = signalling variables

A process/thread stops at a certain point until it is signalled to proceed.

A variable that has an integer value upon which only three operations are defined:

- A semaphore may be initialized to a **nonnegative integer value**
- The `semWait` (or `down` or `P=test`) operation…
  - **decrements** the semaphore value
  - if **value is less than 0** — **block** and **wait for a signal**; else **continue**
- The `semSignal` (or `up` or `V=increment`) operation…
  - **increments** the semaphore value
  - if **value is less than or equal to 0**, ***transmit a signal*** to unblock a waiting thread

```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

See Also: Mutexes vs. Binary Semaphore (vs. General /Counting Semaphores)

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder          */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

-- then check

++ then check

Mutual Exclusion Using Semaphores (NOTE: s=1)

Semaphores

Semaphores = signalling variables
A process/thread stops at a certain point until it is signalled to proceed.

A variable that has an integer value upon which only three operations are defined:

- A semaphore may be initialized to a **nonnegative integer value**
- The `semWait` (or `down` or `P=test`) operation…
  - **decrements** the semaphore value
  - if **value is less than 0** — **block** and **wait for a signal**; else **continue**
- The `semSignal` (or `up` or `V=increment`) operation…
  - **increments** the semaphore value
  - if **value is less than or equal to 0**, ***transmit a signal*** to unblock a waiting thread

Hold waiting processes/threads in a queue…

Strong Semaphore — blocked processes released using queue (FIFO)
Weak Semaphore — order of release is not specified

*Question: What are some **advantages/disadvantages** of strong vs. weak semaphores?*

```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

See Also: Mutexes vs. Binary Semaphore (vs. General /Counting Semaphores)

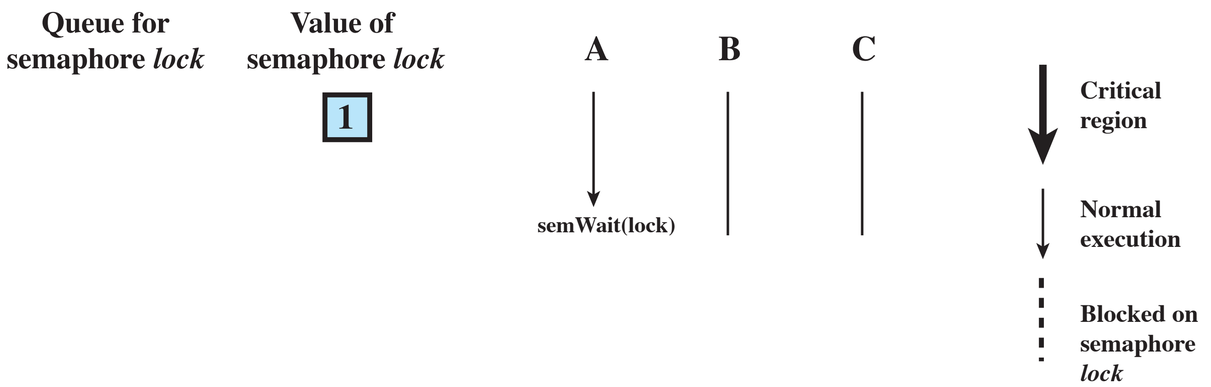
```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder          */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

-- then check

++ then check

Mutual Exclusion Using Semaphores (NOTE: s=1)

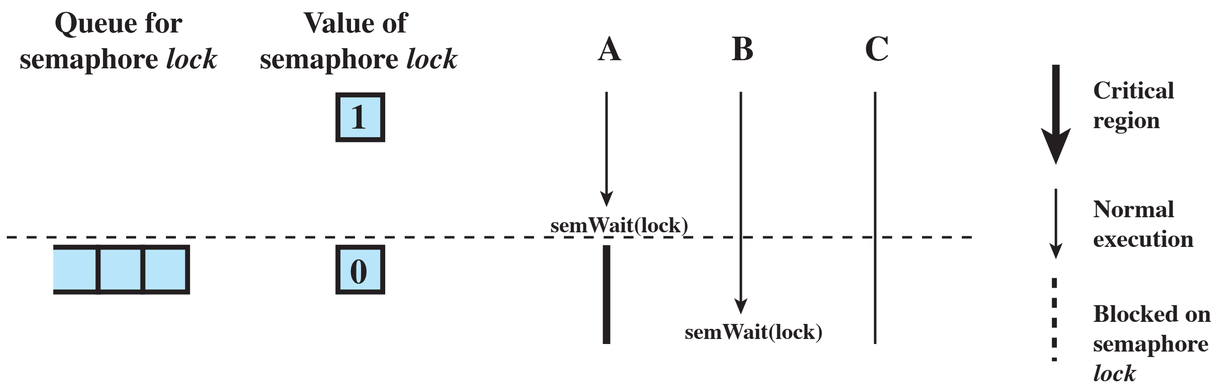
Example: Semaphores In Action




```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

Note that normal execution can proceed in parallel but that critical regions are serialized.

Example: Semaphores In Action

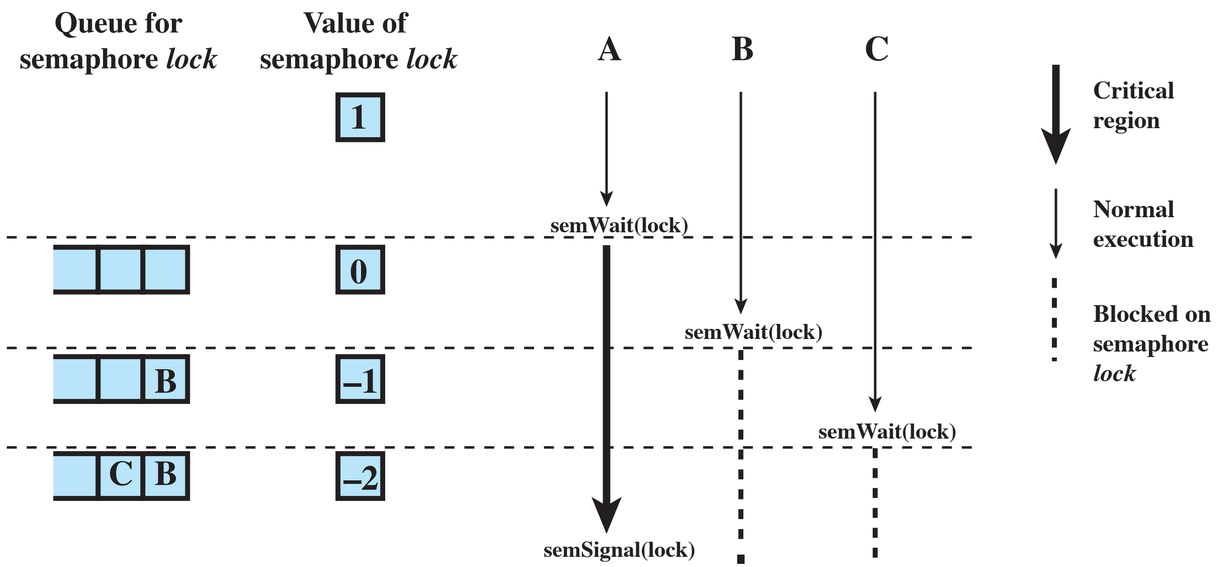



```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

Note that normal execution can proceed in parallel but that critical regions are serialized.

Example: Semaphores In Action

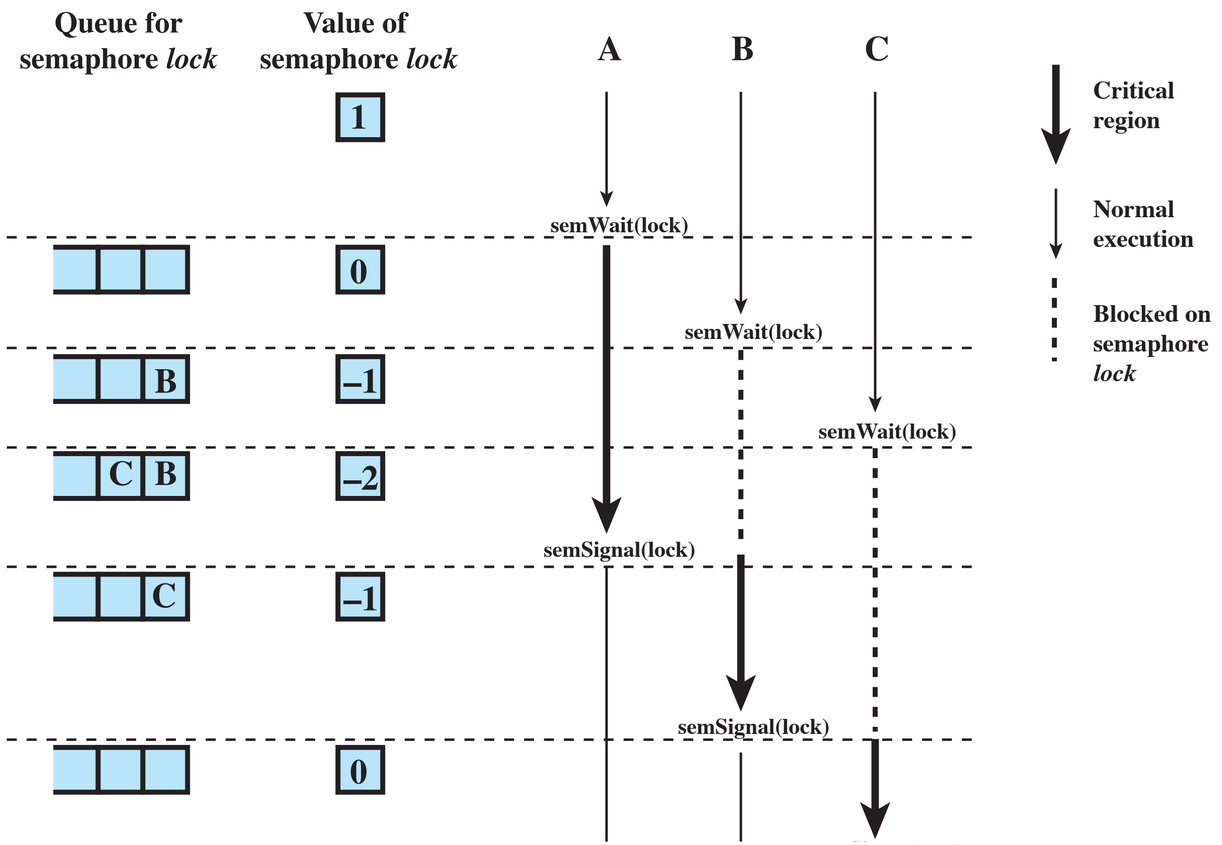
```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```



Note that normal execution can proceed in parallel but that critical regions are serialized.

Example: Semaphores In Action

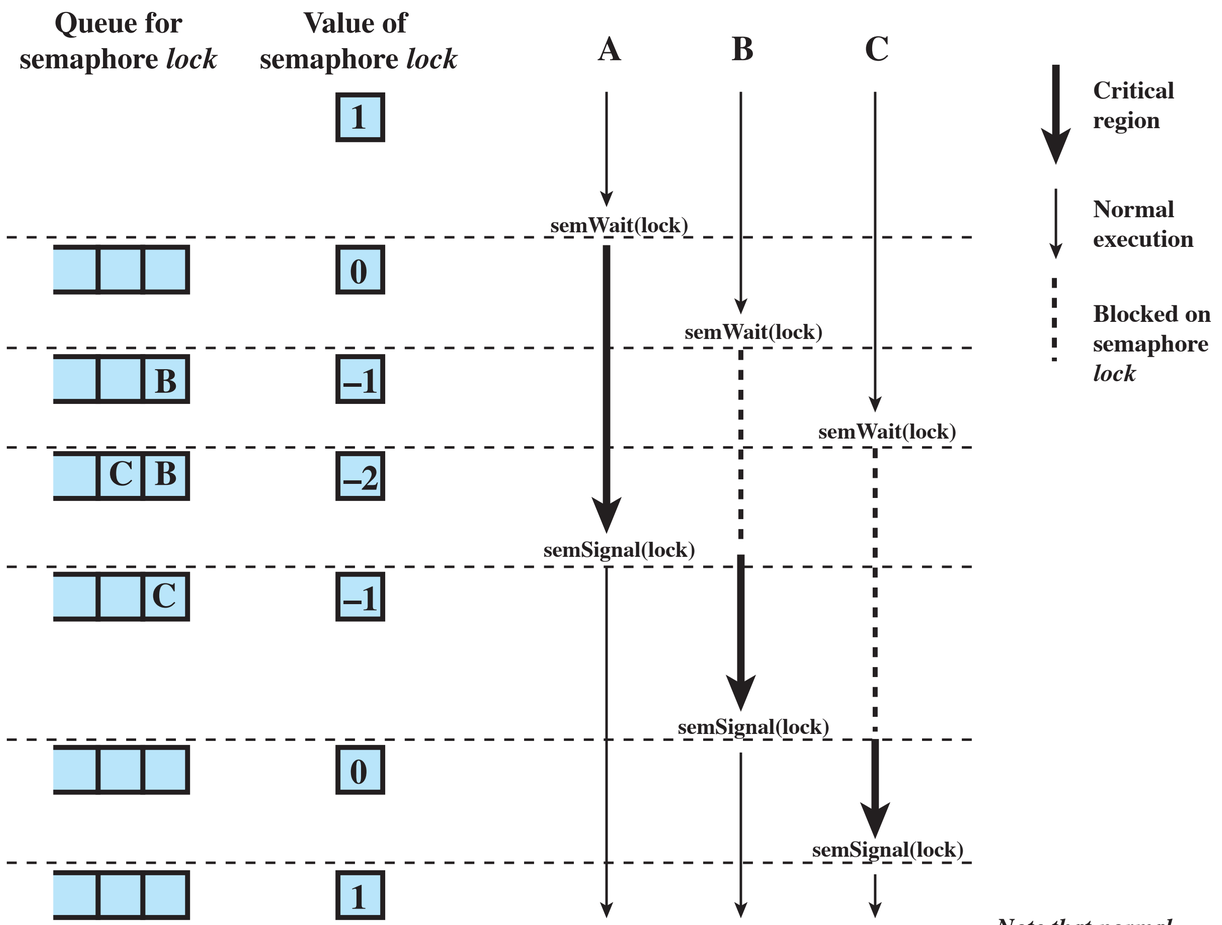
```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```



Note that normal execution can proceed in parallel but that critical regions are serialized.

Example: Semaphores In Action

```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

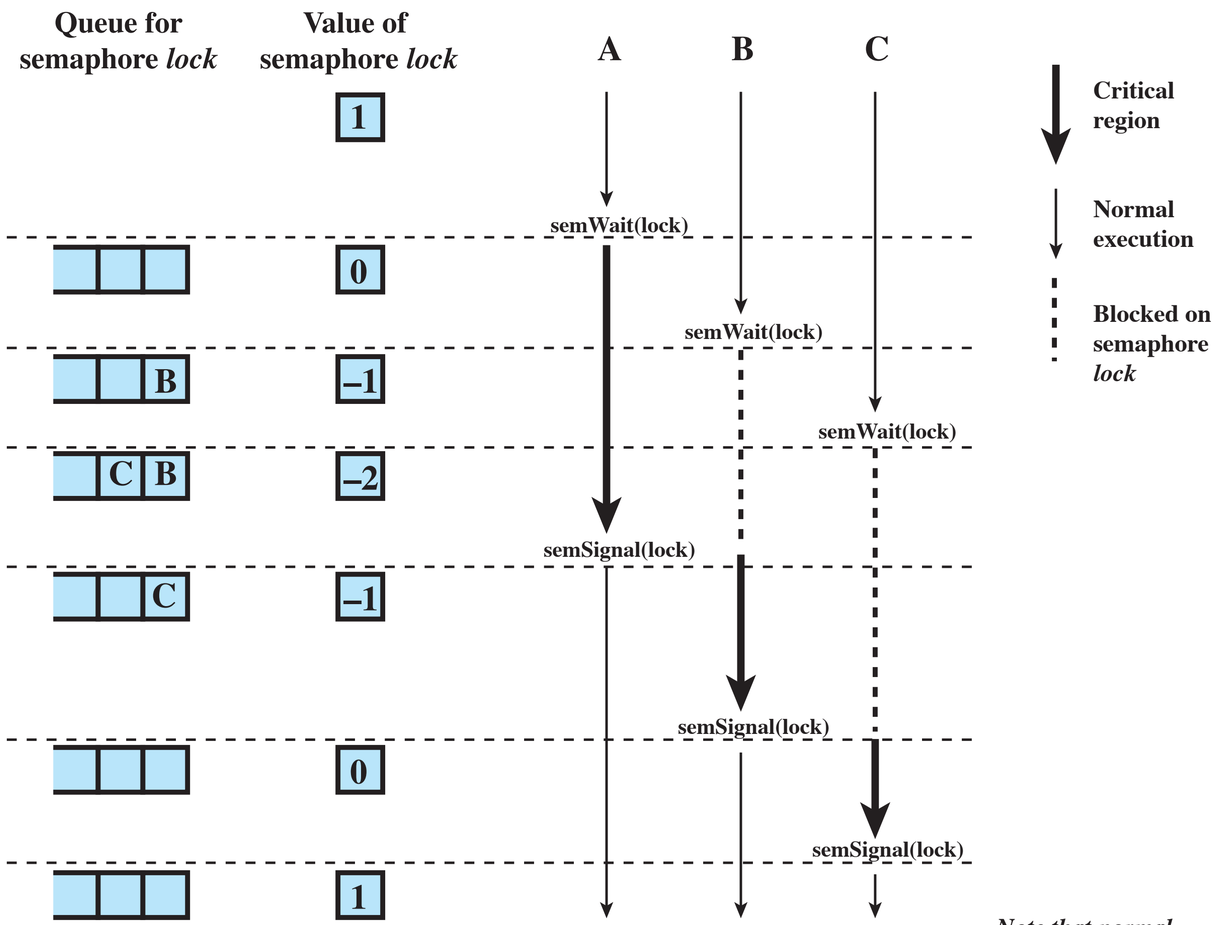


Note that normal execution can proceed in parallel but that critical regions are serialized.

Example: Semaphores In Action

```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

*Question: Is this an example of a **strong semaphore** or a **weak semaphore**? Why?*



Note that normal execution can proceed in parallel but that critical regions are serialized.

Implementing Semaphores

semWait() and semSignal() must be atomic!

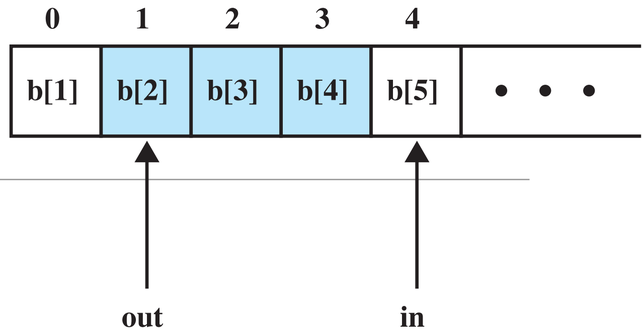
```
semWait(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue*/;
        /* block this process (must also set s.flag to 0)
*/;
    }
    s.flag = 0;
}

semSignal(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
         /* do nothing */;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    s.flag = 0;
}
```

(a) Compare and Swap Instruction

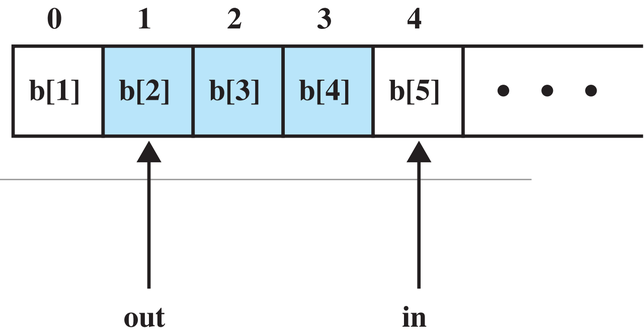
```
semWait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process and allow interrupts */;
    }
    else
       allow interrupts;
}

semSignal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    allow interrupts;
}
```

(b) Interrupts

Producer/Consumer Problem

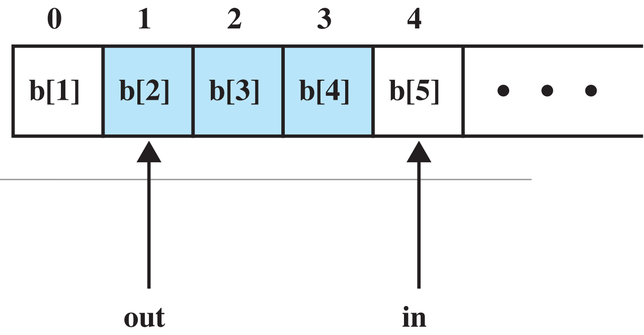


Producer/Consumer Problem



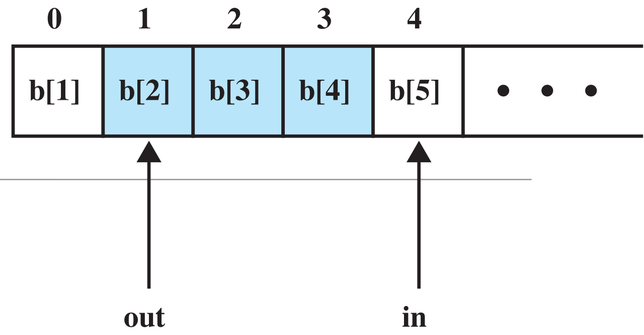
- Buffer shared by a Producer (**P**) and a Consumer (**C**)
  - Producer can produce so long as the there is space to put an item; Consumer can consume so long as there is an item to consume.

Producer/Consumer Problem



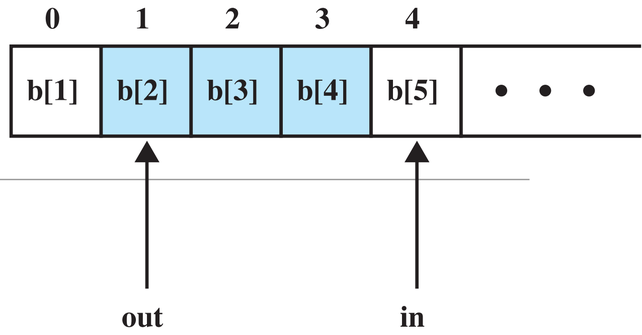
- Buffer shared by a Producer (**P**) and a Consumer (**C**)
  - Producer can produce so long as the there is space to put an item; Consumer can consume so long as there is an item to consume.
- What to do if buffer is ***full***? Or ***empty***?

Producer/Consumer Problem



- Buffer shared by a Producer (**P**) and a Consumer (**C**)
  - Producer can produce so long as the there is space to put an item; Consumer can consume so long as there is an item to consume.
- What to do if buffer is ***full***? Or ***empty***?
- How to alert Producer and Consumer when buffer is ***no longer full or empty***, respectively?

Producer/Consumer Problem



- Buffer shared by a Producer (**P**) and a Consumer (**C**)
  - Producer can produce so long as the there is space to put an item; Consumer can consume so long as there is an item to consume.
- What to do if buffer is ***full***? Or ***empty***?
- How to alert Producer and Consumer when buffer is ***no longer full or empty***, respectively?
- ***See progression in the text (infinite buffer solution, finite buffer, etc.)*** —
  Note discussions of considerations for edge cases with circular buffers, programming errors, etc.

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Infinite Buffer (Incorrect)

```
/* program  producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

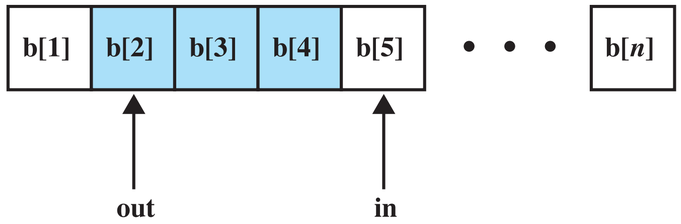
Infinite Buffer (Correct)

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

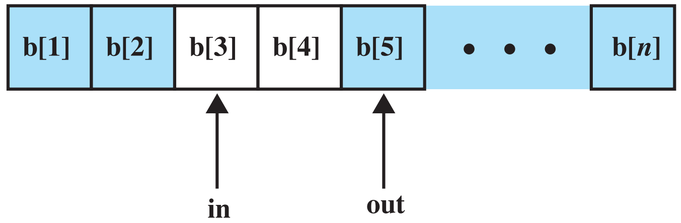
Finite Buffer…

Producer/Consumer Problem (w/ a Bounded Buffer)

- In reality, buffers are not infinite in size… hence, we need a solution for a **Bounded Buffer**.



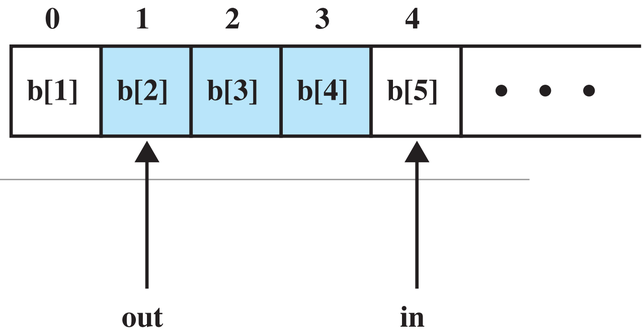
(a)



(b)

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Producer/Consumer Problem



- Buffer shared by a Producer (**P**) and a Consumer (**C**)
  - Producer can produce so long as the there is space to put an item; Consumer can consume so long as there is an item to consume.
- What to do if buffer is ***full***? Or ***empty***?
- How to alert Producer and Consumer when buffer is ***no longer full or empty***, respectively?

```
# Example: Pipes!
wget [URL] mybigfile.txt
# E.g., wget http://www.gutenberg.org/cache/epub/16328/pg16328.txt -O mybigfile.txt

cat mybigfile.txt | tr 'A-Z' 'a-z' | tr -cs 'a-z' '\n' | sort | uniq -c

# vs.
cat mybigfile.txt | tr 'A-Z' 'a-z' | tr -cs 'a-z' '\n' | sort | uniq -c | sort -nr | head -n 20

# vs.
cat | tr 'A-Z' 'a-z'
```

Preview Programming Assignment 1