**Operating Systems**

# "Synchronization and Semaphores"

## Outline

- Fundamentals of semaphore
- Mutual Exclusion Using Semaphores
- Examples
- Analysis
- Implementation

# Semaphores

- Fundamental Principle:
  - Two or more processes want to cooperate by means of simple signals
- Special Variable: **`semaphore s`**
  - A special kind of "int" variable
  - Can't just modify or set or increment or decrement it

# Semaphores

- Before entering critical section
  - **`semWait(s)`**
    - Receive signal via semaphore **`s`**
    - "down" on the semaphore
    - Other term: **`P`** – proberen ("to try, prove")
- After finishing critical section
  - **`semSignal(s)`**
    - Transmit signal via semaphore **`s`**
    - "up" on the semaphore
    - Other term : **`V`** – verhogen (to make or become higher)
- Implementation requirements
  - **`semSignal`** and **`semWait`** must be atomic

# Inside a Semaphore

- Requirement
  - No two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time!
- Critical section
  - **`wait()`** and **`signal()`** code
  - Now have busy waiting in critical section implementation
    - + Implementation code is short
    - + Little busy waiting if critical section rarely occupied
    - – Bad for applications may spend lots of time in critical sections

# Inside a Semaphore

- Add a waiting queue
- Multiple process waiting on **`s`**
  - Wakeup one of the blocked processes upon getting a signal
- Semaphore data structure

```
typedef struct {
  int count;
  queueType queue;
  /* queue for procs. waiting on s */
} SEMAPHORE;
```

# Binary Semaphores

```
typedef struct bsemaphore {
  enum {0,1} value;
  queueType queue;
 } BSEMAPHORE;

void semWaitB(bsemaphore s) {
        if (s.value == 1)
                s.value = 0;
        else {
                place P in s.queue;
                block P;
        }
}

void semSignalB (bsemaphore s) {
  if (s.queue is empty())
        s.value = 1;
  else {
        remove P from s.queue;
        place P on ready list;
  }
}
```

# General Semaphore

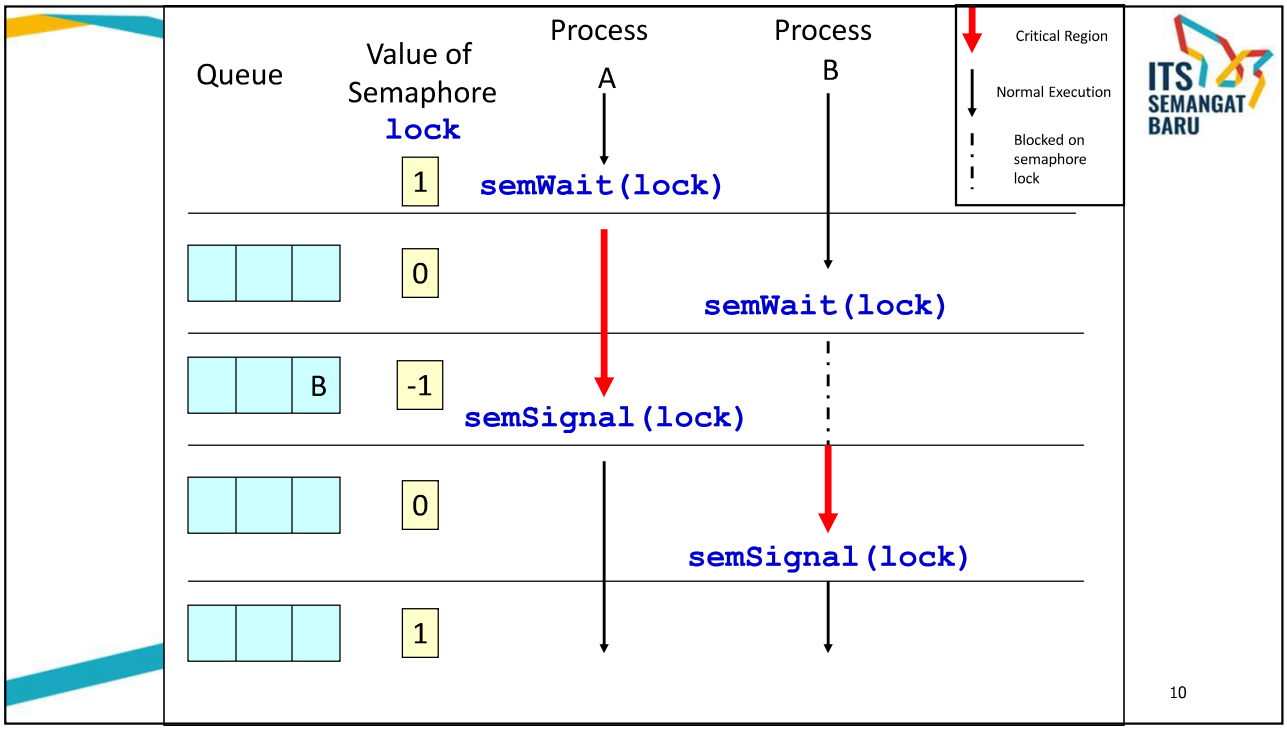
```
typedef struct {
  int count;
   queueType queue;
} SEMAPHORE

void semWait(semaphore s) {
    s.count--;
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }
}

void semSignal(semaphore s) {
  s.count++;
  if (s.count ≤ 0) {
        remove P from s.queue;
        place P on ready list;
  }
}
```

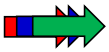
# Mutual Exclusion Using Semaphores

```
semaphore s = 1;
Pi {
 while(1)     {
     semWait(s);
     /* Critical Section */
     semSignal(s);
     /* remainder */
 }
}
```



# Semaphore Example 1

- What happens?
- When might this be desirable?

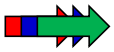


```
semaphore s = 2;
Pi {
  while(1)   {
       semWait(s);
       /* CS */
       semSignal(s);
       /* remainder */
  }
}
```

s = 2 1 0 -1

# Semaphore Example 2

- What happens?
- When might this be desirable?



```
semaphore s = 0;
Pi {
  while(1)   {
       semWait(s);
       /* CS */
       semSignal(s);
       /* remainder */
  }
}
```

s = 0 -1 -2 -3

# Semaphore Example 3

- What happens?
- When might this be desirable?

```
semaphore s = 0;
P1 {
  /* do some stuff */
  semWait(s);
  /* do some more stuff */
}
```

s = ~~0~~ ~~-1~~ 1

# Semaphore Example 4

- Two processes
  - Two semaphores: S and Q
  - Protect two critical variables ‘a’ and ‘b’.
- What happens in the pseudocode if Semaphores S and Q are initialized to 1 (or 0)?

Process 1 executes:

```
while(1) {
      semWait(S);
      a;
      semSignal(Q);
}
```

Process 2 executes:

```
while(1) {
      semWait(Q);
      b;
      semSignal(S);
}
```

# Semaphore Example 4

Process 1 executes:

```
while(1) {
    semWait(S);
    a;
    semSignal(Q);
}
```

Process 2 executes:

```
while(1) {
    semWait(Q);
    b;
    semSignal(S);
}
```

S = ~~0~~ -1

Q = ~~0~~ -1

# Semaphore Example 4

Process 1 executes:

```
while(1) {
    semWait(S);
    a;
    semSignal(Q);
}
```

Process 2 executes:

```
while(1) {
    semWait(Q);
    b;
    semSignal(S);
}
```

S = ~~0~~ ~~-1~~ ~~0~~ 0

Q = ~~0~~ ~~-1~~ ~~0~~ 0

# Semaphore Example 4

Process 1 executes:

```
while(1) {
    semWait(S);
    a;
    semSignal(Q);
}
```

Process 2 executes:

```
while(1) {
    semWait(Q);
    b;
    semSignal(S);
}
```

S = ~~1~~ ~~0~~ ~~-1~~ ~~0~~ 1

Q = ~~1~~ ~~0~~ ~~0~~ 0

# Analysis

Deadlock or Violation of Mutual Exclusion?

1
```
semSignal(s);
critical_section();
semWait(s);
```

2
```
semWait(s);
critical_section();
```

3
```
critical_section();
semSignal(s);
```

4
```
semWait(s);
critical_section();
semWait(s);
```

5
```
semWait(s);
semWait(s);
critical_section();
semSignal(s);
semSignal(s);
```

# Analysis

Deadlock or Violation of Mutual Exclusion?

Mutual exclusion violation

1
```
semSignal(s);
critical_section();
semWait(s);
```

Possible deadlock

2
```
semWait(s);
critical_section();
```

Mutual exclusion violation

3
```
critical_section();
semSignal(s);
```

Certain deadlock!

4
```
semWait(s);
critical_section();
semWait(s);
```

Deadlock again!

5
```
semWait(s);
semWait(s);
critical_section();
semSignal(s);
semSignal(s);
```

# POSIX Semaphores

- Named Semaphores
  - Provides synchronization between related process, between threads and unrelated process
  - Kernel persistence
  - System-wide and limited in number
  - Uses **`sem_open`**
- Unnamed Semaphores
  - Provides synchronization between between related process and between threads
  - Thread-shared or process-shared
  - Uses **`sem_init`**

# Example: bank balance

- Want to shared variable **balance** to be protected by semaphore when used in:
  - **decshared** – a function that decrements the current value of **balance**
  - **incshared** – a function that increments the **balance** variable.

# Example: bank balance

```
int decshared() {
   while (sem_wait(&balance_sem) == -1)
      if (errno != EINTR)
         return -1;
   balance--;
   return sem_signal(&balance_sem);
}

int incshared() {
   while (sem_wait(&balance_sem) == -1)
      if (errno != EINTR)
         return -1;
   balance++;
   return sem_signal(&balance_sem);
}
```

# Thank You