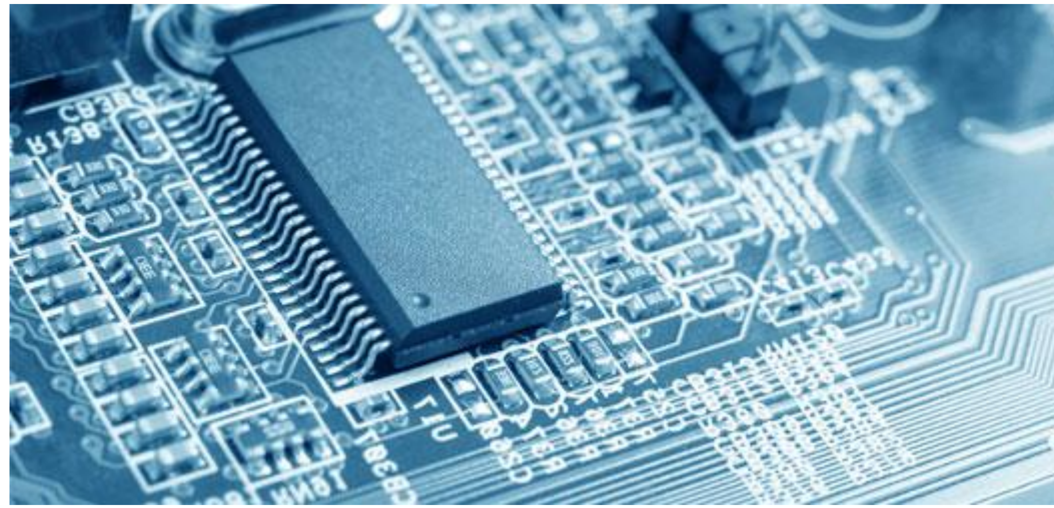
NAZARBAYEV UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY



ROBT305 - Embedded Systems

Lecture 6 – Process Synchronization: Semaphores

8 November, 2015

Course Logistics

Reading Assignment: **Chapters 3, 4, 6** of the of the Operating Systems Concept textbook (relevant material only)
Chapter 3 of the Real-Time Systems Design and Analysis textbook (relevant material only)
Homework Assignment #1 is out in Moodle and due to end of 13 September (Sunday)
Quiz #2 is on 17 September – Pthreads, Mutexes

Semaphores

- The most common method for protecting critical regions involves a special variable called a **semaphore**.
- A semaphore ***S*** is a specific memory location (integer variable) that acts as a lock to protect critical regions.
- Two system calls, **wait** and **signal** are used either to take or to release the semaphore.
- Traditionally, one denotes the **wait** operation as **P** (from the Dutch ***proberen,*** "to test") and the **signal** operations **V** (from ***verhogen,*** "to increment").

The only difference between a "mutex" and a semaphore is that the mutex has to be unlocked by the same thread that locked it.

Semaphores

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1

```
wait (S) {
     while (S <= 0)
          ; // busy wait
     S--;
}

signal (S) {
     S++;
}
```

- Consider tasks P_1 and P_2 that require statement S_1 to happen before statement S_2

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- A common semaphore **Synch** is initialized to 0
- P_2 will execute S_2 only after P_1 has invoked **signal(synch)** after statement S_1

Pthreads Synchronization

- Semaphores belong to the POSIX SEM extension
- POSIX specifies two types of semaphores: **named** and **unnamed**

The code below illustrates the sem_init() function for creating and initializing an unnamed semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);

/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

The **`sem_init()`** *function has three parameters:*

1. *A pointer to the semaphore*
2. *A flag indicating the level of sharing*
3. *The semaphore's initial value*

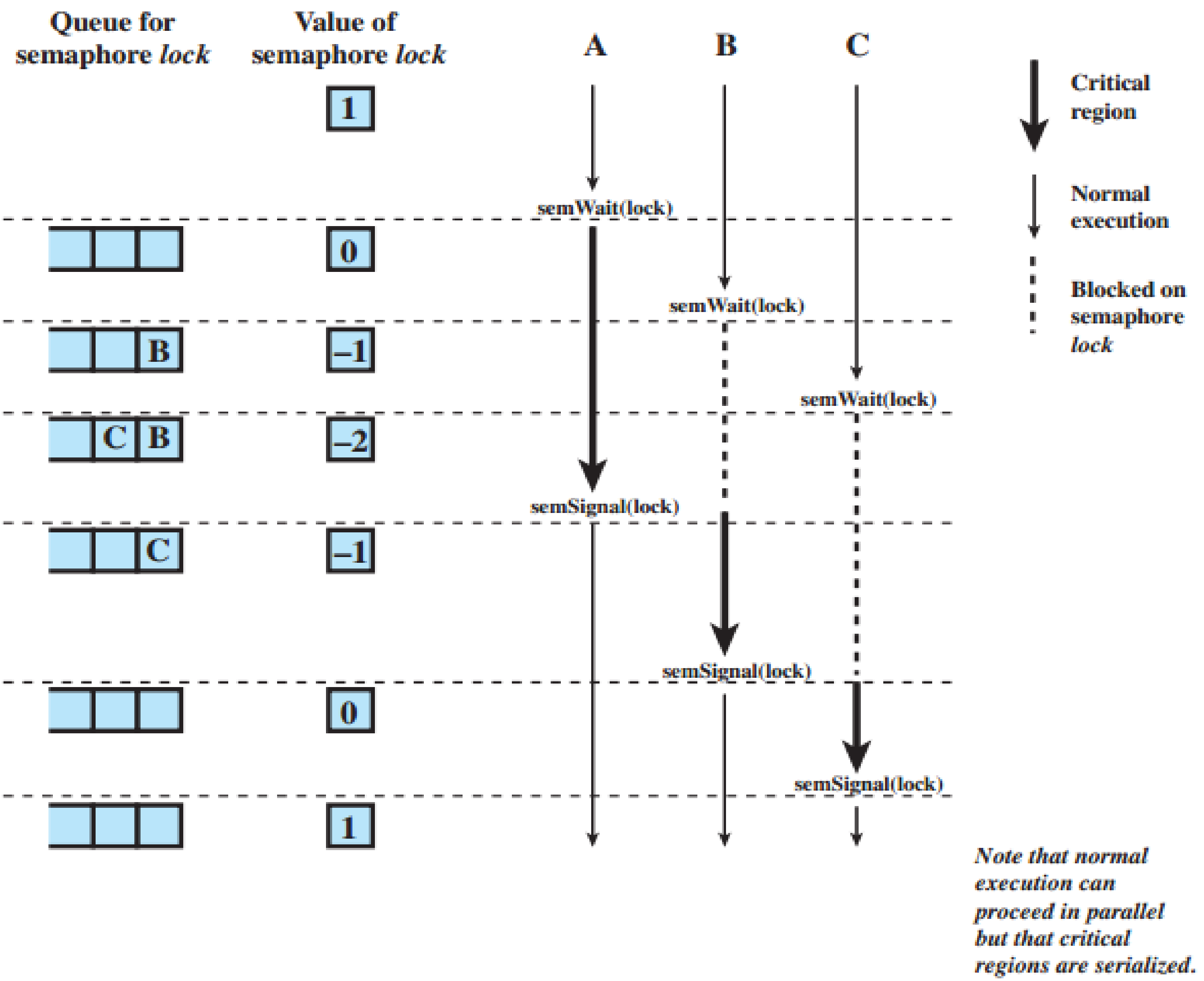
Binary Semaphore Example:
Serially Reusable Resource

- Preemptive priority embedded system with separate channels for acceleration (Task 1) and temperature periodic measurements (Task 2) and a single A/D converter.
- For A/D conversion, the desired channel must be selected.
- Task 1 -High priority; Task 2 - low priority

```
/* Task_1 */
...
wait(&s);  /* wait until A/D available */
select_channel(acceleration);
a_data=ad_conversion(); /* measure */
signal(&s) /* release A/D */
...

/* Task_2 */
...
wait(&s);  /* wait until A/D available */
select_channel(temperature);
t_data=ad_conversion(); /* measure */
signal(&s) /* release A/D */
...
```

Counting Semaphores



Bounded-Buffer (Ring Buffer) Problem

- ***n*** buffers, each can hold one item
- Semaphore **`S`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value **n**

Bounded-Buffer Problem

- The structure of the **producer** task

```
do {
      ...
   /* produce an item in next_produced */
      ...
   wait(empty);
   wait(S);
      ...
   /* add next produced to the buffer */
      ...
   signal(S);
   signal(full);
} while (true);
```

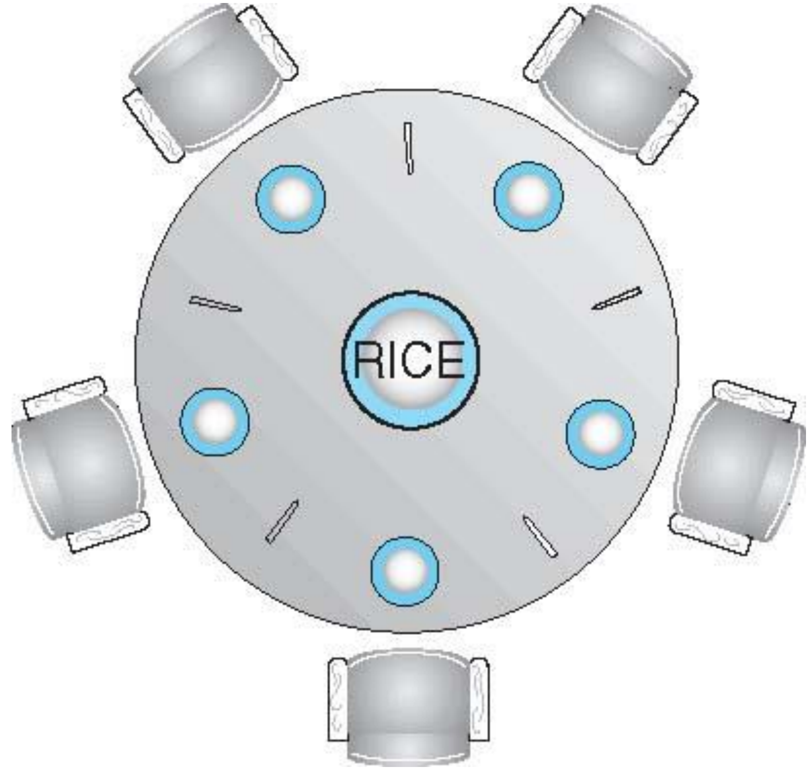
Bounded-Buffer Problem (Cont.)

- The structure of the **consumer** task

```
do {
   wait(full);
   wait(S);
      ...
   /* remove an item from buffer to next_consumed
*/
      ...
   signal(S);
   signal(empty);
      ...
   /* consume the item in next consumed */
      ...
} while (true);
```

Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - 1 bowl of rice (data set)
    - 5 chopstick available
    - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do  {
        wait (chopstick[i] );
        wait (chopstick[ (i + 1) % 5] );

             //  eat

        signal (chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

              //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
room = 4;
do  {
        wait (room);
        wait (chopstick[i] );
         wait (chopStick[ (i + 1) % 5] );

                    //  eat

         signal (chopstick[i] );
         signal (chopstick[ (i + 1) % 5] );
         signal (room);
                      //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation

Any Questions?