Process Synchronization (4)

Dr. Jun Zheng

CSE325 Principles of Operating Systems

10/4/2019

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

In-Class Work 3

- The following figure shows the sequence of semaphore operations at the beginning and at the end of the tasks A, B.
- Determine for each of the 4 cases a, b, c and d, whether or in which sequence the tasks are executed, using the initializations of the semaphore variables given in Table 1.
- wait() is an atomic operation that waits for semaphore to become positive, then decrements it by 1, i.e. if the semaphore is 0, wait() is blocked.
- signal() is an atomic operation that increments the semaphore by 1, waking up a waiting wait() if any.
- SA, SB are the two semaphores corresponding to Tasks A and B.

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

In-Class Work 3

Task A	Task B
wait(SA)	wait(SB)
wait(SA)	wait(SB)
…	…
…	…
…	…
signal(SB)	signal(SA)
END	END

Table 1

	a	b	c	d
SA	2	1	0	2
SB	0	1	2	1

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Answer

a: task A

b: No task will be executed

c: task B

d: task A, task B

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Bounded-Buffer Problem

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

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Bounded Buffer Problem (Cont.)

The structure of the producer process

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

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Bounded Buffer Problem (Cont.)

The structure of the consumer process

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

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Readers-Writers Problem (Cont.)

The structure of a writer process

```
do {
    wait(rw_mutex);
                ...
      /* writing is performed */
                ...
    signal(rw_mutex);
} while (true);
```

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

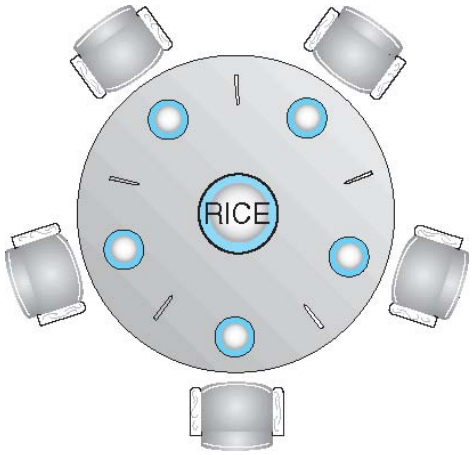
Readers-Writers Problem (Cont.)

The structure of a reader process

```
do {
    wait(mutex);
    read_count++;       /* Protected by mutex */
    if (read_count == 1) /* First reader in */
       wait(rw_mutex); /* Lock out writers */
    signal(mutex);
        ...
        /* reading is performed */
        ...
    wait(mutex);
    read count--;  /* Protected by mutex */
    if (read_count == 0)  /* Last out */
            signal(rw_mutex); /* Let in writers */
    signal(mutex);
} while (true);
```

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Dining-Philosophers Problem



- Philosophers spend their lives alternating 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
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

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?

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY