Process Synchronization and Semaphores

Tahira Alam

University of Asia Pacific

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Consumer-Producer problem

- A producer process produces information that is consumed by a consumer process.
- One solution: producer and consumer use shared memory.
  - allow producer and consumer processes to run concurrently
  - available a buffer of items that can be filled by the producer and emptied by the consumer.
  - The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Bounded and Unbounded Buffer

- The unbounded buffer places no practical limit on the size of the buffer
- The bounded buffer assumes a fixed buffer size.

Solution (Producer)

```
while (count == BUFFER.SIZE)
   ; // do nothing

// add an item to the buffer
buffer[in] = item;
in = (in + 1) % BUFFER.SIZE;
++count;
```

Solution(Consumer)

```
while (count == 0)
   ; // do nothing

// remove an item from the
buffer item = buffer[out];
out = (out + 1) % BUFFER.SIZE;
--count;
```

Solution(Explanation)

The shared buffer is a circular array. ‘in’ points to the next free position, ‘out’ points to the first full position. ‘count’ is the number of elements currently in the buffet. When, count==0, that means buffet is empty and when count == buffet size, that means buffet is full. The array is a **circular array**.

Problem????

- Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently.

$$register_1 = \texttt{count}$$
$$register_1 = register_1 + 1$$
$$\texttt{count} = register_1$$

$$register_2 = \texttt{count}$$
$$register_2 = register_2 - 1$$
$$\texttt{count} = register_2$$

So…..What can happen?

T_0:	*producer*	execute	$register_1 = \texttt{count}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{count}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{count} = register_1$	$\{count = 6\}$
T_5:	*consumer*	execute	$\texttt{count} = register_2$	$\{count = 4\}$

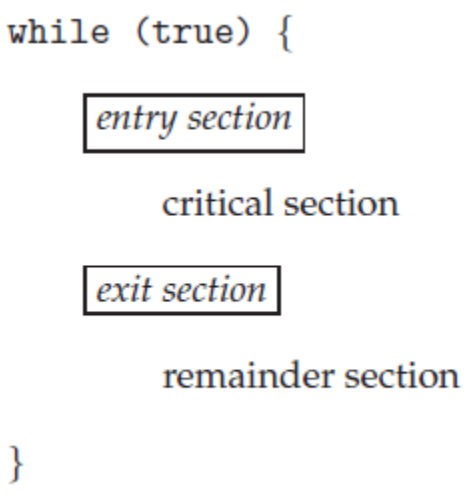
Race Condition

- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition.**

The Critical-Section Problem

- Consider a system consisting of *n processes {P0, P1, ..., Pn−1}. Each process* has a segment of code, called a **critical section,** in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

General Structure of a Code Having Critical Section



Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section.**

Requirements for Solution for a Critical-Section Problem

- Mutual Exclusion
- Progress
- Bounded Waiting

Paterson's Solution to Bounded Buffer System

- Applicable for Two Processes
- Suppose Process **i** and **j (numeric values: 0,1)**

```
int turn;
boolean flag[2];

while (true) {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

}
```

Hardware Solution

- Well there are some hardware solutions, but they are too much complicated.
- No worries, we have Semaphore!!!!!

Semaphore

- Synchronization primitive like condition variables
- Semaphore is a variable with an underlying counter
- Two functions on a semaphore variable
  - Up/post increments the counter
  - Down/wait decrements the counter and blocks the calling thread if the resulting value is negative
- A semaphore with init value 1 acts as a simple lock (binary semaphore = mutex)

Semaphore Example

```
sem_t m;
sem_init(&m, 0,1);//initialize semaphore to X; what should X be?
sem_wait(&m);
// critical section here
sem_post(&m);
```

Solving the Producer-consumer Problem using Semaphores

- Need two semaphores for signaling
  - One to track empty slots, and make producer wait if no more empty slots
  - One to track full slots, and make consumer wait if no more full slots
  - One semaphore to act as mutex for buffer

Solving the Producer-consumer Problem using Semaphores(2)

```
int main(int argc, char *argv[]) {
 // ...
 sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with ...
 sem_init(&full, 0, 0); // ... and 0 are full
 sem_init(&mutex, 0, 1); // mutex=1 because it is a lock
 // ...
}
```

Solving the Producer-consumer Problem using Semaphores(3)

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
 int i;

 for (i = 0; i < loops; i++) {
   sem_wait(&empty);
   sem_wait(&mutex);
   put(i);
   sem_post(&mutex);
   sem_post(&full);
 }
}
```

```
void *consumer(void *arg) {
 int i;

 for (i = 0; i < loops; i++) {
   sem_wait(&full);
   sem_wait(&mutex);
   int_ tmp = get();
   sem_post(&mutex);
   sem_post(&empty);
   printf("%d\n", tmp);
 }
}
```

Reader Writer Problem

- One database
- Some processes want to read only (**Readers**)
- Some processes want to read and write (**Writers**)
- Problem:
  - Reader and Reader→ok
  - Reader and writer → not ok
  - Writer and reader → not ok
  - Writer and Writer → not ok

Writer

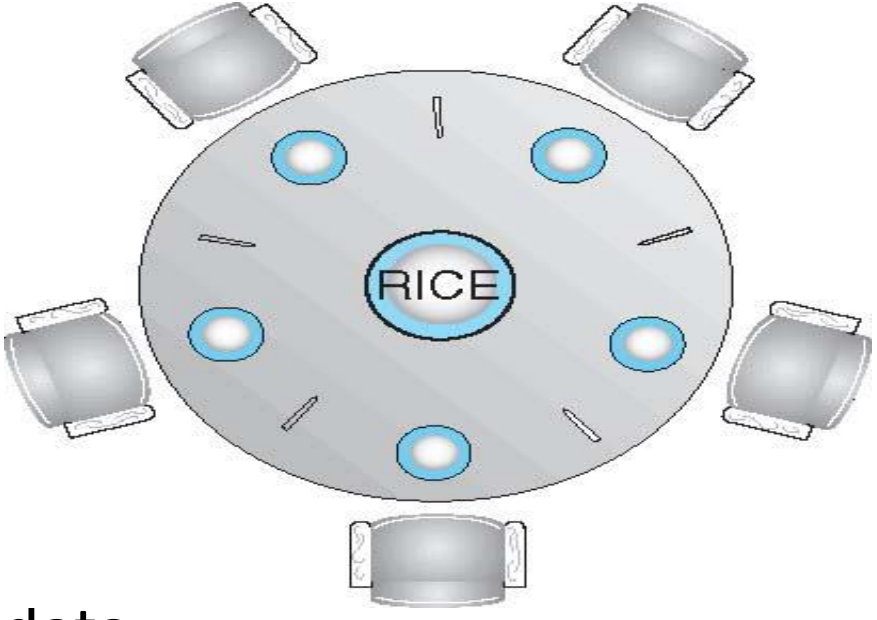
```
semaphore mutex, wrt;
int readcount;
```

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

Reader

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
} while (TRUE);
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set) and need 2 chopsticks
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

```
semaphore chopstick[5];

do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
} while (TRUE);
```

Thank you!!!!