ILAHIA COLLEGE OF ENGINEERING AND TECHNOLOGY

OPERATING SYSTEM-S4 CSE

Module III

Process Synchronization: Critical SectionPeterson's solution. Synchronization – Locks, Semaphores, Monitors, Classical Problems – Producer Consumer, Dining Philosophers and Readers-Writers Problems

Process Synchronization

A **co-operating process** is one that can affect or be affected by other processes executing in the system. Cooperating process may either directly share a logical address space (that is, both code and data) or be allowed to share data only through files. The former case is achieved through the use of lightweight processes or threads.

Concurrent access to shared data may result in data inconsistency. 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**. To guard against the race condition we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require some form of synchronization of the processes. Such situations occur frequently in operating system systems as different parts of the system manipulate resources and we want the changes not to interfere with one another. A major portion of this module is concerned with process synchronization and coordination

- we looked at cooperating processes (those that can effect or be effected by other simultaneously running processes), and as an example, we used the producer-consumer cooperating processes:

  Producer code :

```
item nextProduced;

while( true ) {

/* Produce an item and store it in nextProduced */
nextProduced = makeNewItem( . . . );

/* Wait for space to become available */
while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
    ; /* Do nothing */

/* And then store the item and repeat the loop. */
```

```
buffer[ in ] = nextProduced;
in = ( in + 1 ) % BUFFER_SIZE;

}
```

Consumer code :

```
item nextConsumed;

while( true ) {

/* Wait for an item to become available */
while( in == out )
    ; /* Do nothing */

/* Get the next available item */
nextConsumed = buffer[ out ];
out = ( out + 1 ) % BUFFER_SIZE;

/* Consume the item in nextConsumed
   ( Do something with it ) */

}
```

- The only problem with the above code is that the maximum number of items which can be placed into the buffer is BUFFER_SIZE - 1. One slot is unavailable because there always has to be a gap between the producer and the consumer.
- We could try to overcome this deficiency by introducing a counter variable, as shown in the following code segments:

Producer Process:

```
while (true)
{
     /* produce an item in nextProduced */
     while (counter == BUFFER_SIZE)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Consumer Process:

```
while (true)
{
     while (counter == 0)
        ; /* do nothing */
     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;
     /* consume the item in nextConsumed */
}
```

- Unfortunately we have now introduced a new problem, because both the producer and the consumer are adjusting the value of the variable counter, which can lead to a condition known as a ***race condition***. In this condition a piece of code may or may not work correctly, depending on which of two simultaneous processes executes first, and more importantly if one of the processes gets interrupted such that the other process runs between important steps of the first process. (Bank balance example discussed in class.)
- The particular problem above comes from the producer executing "counter++" at the same time the consumer is executing "counter--". If one process gets part way through making the update and then the other process butts in, the value of counter can get left in an incorrect state.
- But, you might say, "Each of those are single instructions - How can they get interrupted halfway through?" The answer is that although they are single instructions in C++, they are actually three steps each at the hardware level: (1) Fetch counter from memory into a register, (2) increment or decrement the register, and (3) Store the new value of counter back to

memory. If the instructions from the two processes get interleaved, there could be serious problems, such as illustrated by the following:

Producer:

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

Consumer:

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

Interleaving:

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

- Note that race conditions are ***notoriously difficult*** to identify and debug, because by their very nature they only occur on rare occasions, and only when the timing is just exactly right. (or wrong! :-)) Race conditions are also very difficult to reproduce. :-(
- Obviously the solution is to only allow one process at a time to manipulate the value "counter". This is a very common occurrence among cooperating processes, so lets look at some ways in which this is done, as well as some classic problems in this area.

The Critical-Section Problem

- Consider system of *n* processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
  - *Critical section problem* is to design protocol to solve this

- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

The general structure of a typical process Pi having critical section is as shown below.

```
do {
        Entry Section
    Critical section
        Exit section
    Remainder section
} while (1);
```

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual Exclusion** - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress** - If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section.
3. **Bound Waiting** - There exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- Two approaches depending on if kernel is preemptive or non-preemptive
    - **Preemptive** – allows preemption of process when running in kernel mode
    - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
        - Essentially free of race conditions in kernel mode

Peterson's Solution

It is the classic software based solution to the critical section problem.It is restricted to two processes. The processes are numbered p_i and p_j.

- The two processes share two variables:
    - **int turn;**

- **Boolean flag[2]**

- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = true** implies that process $\mathbf{P_i}$ is ready!
- The structure of process in algorithm 3 is as shown below.

```
Do {

    Flag[i] = true;

    Turn =j;

    While (flag[j] && turn ==j);

        Critical section

    Flag[i] = false;

        Remainder section

} while (1);
```

- **Provable that**
  - **Mutual exclusion is preserved**
  - **Progress requirement is satisfied**
  - **Bounded-waiting requirement is met**

Synchronization hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of locking
  - Protecting critical regions via locks
  - Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
    - Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptible
  - Either test memory word and set value

- Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = TRUE;
        return rv:
}
```

Solution using test_and_set()

- Shared boolean variable lock, initialized to FALSE
- Solution:

```
do {
  while (test_and_set(&lock))
; /* do nothing */
/* critical section */
```

```
lock = false;

/* remainder section */

} while (true);
```

compare_and_swap Instruction

Definition:

```
int compare and swap(int *value, int expected, int new_value) {
  int temp = *value;
  if (*value == expected)
    *value = new_value;
  return temp;
}
```

Solution using compare_and_swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

```
do {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ; /* do nothing */
    /* critical section */
  lock = 0;
    /* remainder section */
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
  waiting[i] = true;
```

```
        key = true;
        while (waiting[i] && key)
          key = test_and_set(&lock);
        waiting[i] = false;
        /* critical section */
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
          j = (j + 1) % n;
        if (j == i)
          lock = false;
        else
          waiting[j] = false;
        /* remainder section */
      } while (true);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Product critical regions with it by first acquire() a lock then release() it
  - Boolean variable indicating if lock is available or not
- Calls to acquire() and release() must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires busy waiting
    - This lock therefore called a spinlock

acquire() and release()

```
acquire() {
  while (!available)
    ; /* busy wait */
  available = false;;
}
release() {
  available = true;
}
do {
  acquire lock
    critical section
  release lock
    remainder section
} while (true);
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore ***S*** – integer variable
- Two standard operations modify ***S***: **wait()** and **signal()**
  - Originally called **P()** and **V()**
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

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

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

Semaphore Usage

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1
  - Then a mutex lock
- Can implement a counting semaphore *S* as a binary semaphore
- Can solve various synchronization problems
- Consider two processes P_1 with a statement S_1 and P_2 with a statement S_2 Suppose we require that S_2 to be executed only after S_1 has completed.We can implement this scheme by letting p1 and p2 share a common semaphore synch,initialized to 0 and by inserting statements

  S_1;

  signal(synch);

  in process in P1 and statements

  wait(synch);

  S_2;

  in P2

Mutual exclusion implementation with semaphores

```
Semaphore mutex;    //  initialized to 1
do {
        wait (mutex);
// Critical Section
signal (mutex);
```

// remainder section

} while (TRUE);

Semaphore Implementation

The main disadvantages of the semaphore definition given here is that it requires **busy waiting.** While a process is in critical section**,**any other process that tries to enter its critical section must loop continuously in the entry code.This type of semaphore is called a **spinlock** because the process spins while waiting for the lock.

To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations.When a process execute the wait() operation and finds that the semaphore value is not positive, it must wait.Rather than engaging in busy waiting, the process can block itself. **Block** operation places a process into a waiting queue associated with the semaphore,and the state of the process is switched to the waiting state.A process that is blocked should be restart ed by a wake up() operation. **wakeup** – operation remove one of processes in the waiting queue and place it in the ready queue.Its state changes from waiting to ready.

We define a semaphore as

```
typedef struct{
  int value;
  struct process *list;
} semaphore;
```

Each semaphore has two data items:

- **value** (of type integer)
- List of processes **list**

When a process must wait on a semaphore,it is added to the list of processes.A signal() operation remove one process from the list of waiting processes aand awakens that process

Implementation of wait:

```
wait(semaphore *S) {
    S->value--;
```

```
        if (S->value < 0) {
                add this process to S->list;
                block();
                }
        }
```

Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
                }
        }
```

Deadlock and Starvation

Several processes compete for a finite set of resources in a multiprogrammed environment. A process requests for resources that may not be readily available at the time of the request. In such a case the process goes into a wait state. It may so happen that this process may never change state because the requested resources are held by other processes which themselves are waiting for additional resources and hence in a wait state. This situation is called a deadlock.

two (or more) processes are waiting for an event that can be caused by only one of the waiting processes, e.g. let S and Q be two semaphores initialized to 1:

```
P0              P1
wait (S);       wait (Q);
wait (Q);       wait (S);
  :               :
signal  (S);    signal (Q);
```

signal (Q); signal (S);

Suppose P0 executes wait(S) and then P1 executes wait(Q). When P0 executes wait(Q),it must wait until P1 executes signal(Q).Similarly, when P1 executes wait(S),it must wait until P0 executes sugnal(S).Since these signal() operations cannot be executed,P0 and P1 are deadlocked.

Another problem related to deadlocks is **indefinite blocking or starvation**, a situation in which processes wait indefinetly within the semaphore.

- **Priority inversion** - A challenging scheduling problem arises when a high-priority process gets blocked waiting for a resource that is currently held by a low-priority process.
- If the low-priority process gets pre-empted by one or more medium-priority processes, then the high-priority process is essentially made to wait for the medium priority processes to finish before the low-priority process can release the needed resource, causing a ***priority inversion.*** If there are enough medium-priority processes, then the high-priority process may be forced to wait for a very long time.
- One solution is a ***priority-inheritance protocol,*** in which a low-priority process holding a resource for which a high-priority process is waiting will temporarily inherit the high priority from the waiting process. This prevents the medium-priority processes from preempting the low-priority process until it releases the resource, blocking the priority inversion problem.

Classic Problems of Synchronization

- **Bounded-Buffer Problem**
- **Readers and Writers Problem**
- **Dining-Philosophers Problem**

Bounded-Buffer Problem

- This is a generalization of the producer-consumer problem wherein access is controlled to a shared group of buffers of a limited size.
- In this solution, the two counting semaphores "full" and "empty" keep track of the current number of full and empty buffers respectively (and initialized to 0 and N respectively.) The binary semaphore mutex controls access to the critical section. The producer and consumer processes are nearly identical - One can think of the producer as producing full buffers, and the consumer producing empty buffers.

```
do {
    . . .
   // produce an item in nextp
    . . .
   wait(empty);
   wait(mutex);
    . . .
   // add nextp to buffer
    . . .
   signal(mutex);
   signal(full);
}while (TRUE);
```

Figure 5.9 The structure of the producer process.

```
do {
   wait(full);
   wait(mutex);
    . . .
   // remove an item from buffer to nextc
    . . .
   signal(mutex);
   signal(empty);
    . . .
   // consume the item in nextc
    . . .
}while (TRUE);
```

Figure 5.10 The structure of the consumer process.

The Readers-Writers Problem

A data set is shared among a number of concurrent processes

- readers – only read the data set
- writers – can both read and write

We can allow multiple readers to read at the same time. However, only one single writer can access the shared data at a time.

There are several variations to the readers-writers problem, most centered around relative priorities of readers versus writers.

- The *first* readers-writers problem gives priority to readers. In this problem, if a reader wants access to the data, and there is not already a writer accessing it, then access is granted to the reader. A solution to this problem can lead to starvation of the writers, as there could always be more readers coming along to access the data.
- The *second* readers-writers problem gives priority to the writers. In this problem, when a writer wants access to the data it jumps to the head of the queue - All waiting readers are blocked, and the writer gets access to the data as soon as it becomes available. In this solution the readers may be starved by a steady stream of writers.

The following code is an example of the first readers-writers problem, and involves an important counter and two binary semaphores:

- *readcount* is used by the reader processes, to count the number of readers currently accessing the data.
- *mutex* is a semaphore used only by the readers for controlled access to readcount.
- *wrt* is a semaphore used to block and release the writers. The first reader to access the data will set this lock and the last reader to exit will release it; The remaining readers do not touchrw_mutex
- Note that the first reader to come along will block on wrt if there is currently a writer accessing the data, and that all following readers will only block on mutex for their turn to increment readcount.

➢ The structure of a Writer process

```
do {

wait (wrt) ;

//    writing is performed

signal (wrt) ;

} while (TRUE);
```

- The structure of a Reader process

```
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);
```

3. Dining Philosophers Problem

- The dining philosophers problem is a classic synchronization problem involving the allocation of limited resources amongst a group of processes in a deadlock-free and starvation-free manner:

Consider five philosophers sitting around a table, in which there are five chopsticks evenly distributed and an endless bowl of rice in the center, as shown in the diagram below. (There is exactly one chopstick between each pair of dining philosophers.)These philosophers spend their lives alternating between two activities: eating and thinking.When it is time for a philosopher to eat,

it must first acquire two chopsticks - one from their left and one from their right.When a philosopher thinks, it puts down both chopsticks in their original locations.

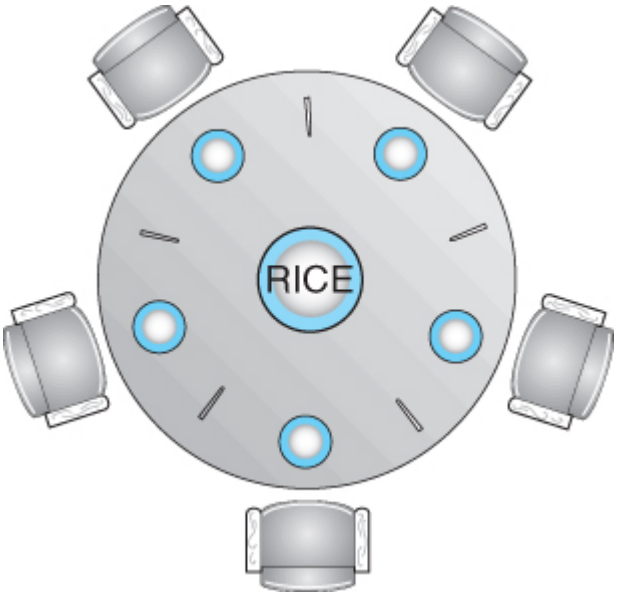

Figure 5.13 - The situation of the dining philosophers

- One possible solution, as shown in the following code section, is to represent each chopstick with a semaphore.A philosopher tries to grab a chopstick by executing a wait() operation and release her chopstick by executing the signal() operation. use a set of five semaphores (chopsticks[5]), and to have each hungry philosopher first wait on their left chopstick (chopsticks[i]), and then wait on their right chopstick (chopsticks[(i + 1) % 5])
- But suppose that all five philosophers get hungry at the same time, and each starts by picking up their left chopstick. They then look for their right chopstick, but because it is unavailable, they wait for it, forever, and eventually all the philosophers starve due to the resulting deadlock.

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

Figure 5.14 - The structure of philosopher i.

- Some potential solutions to the problem include:
    - Only allow four philosophers to dine at the same time. (Limited simultaneous processes.)
    - Allow philosophers to pick up chopsticks only when both are available, in a critical section. (All or nothing allocation of critical resources.)
    - Use an asymmetric solution, in which odd philosophers pick up their left chopstick first and even philosophers pick up their right chopstick first5.8 Monitors