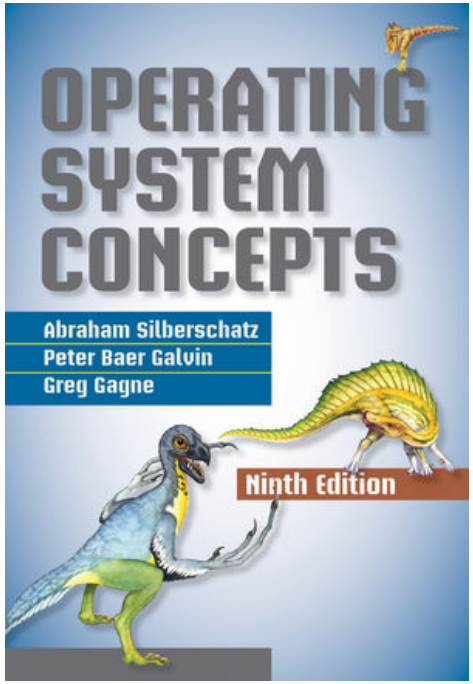
4 - Process Synchronization

EECE 315 (101)
ECE – UBC
2013 W2



Acknowledgement: This set of slides is partly based on the PPTs provided by the Wiley's companion website (including textbook images, when not explicitly mentioned/referenced).

Lecture Outline

- **Background**
  - The Critical-Section Problem
  - Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization

Introduction

- As discussed earlier, cooperating processes can either directly share a logical address or be allowed to share data through files or messages.
  - Concurrent access to shared data may result in data inconsistency
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- To describe the problem, let's look at an example:
  - Consider the consumer-producer problem that we discussed in Ch 3
    - Suppose that we want to provide a solution to the consumer-producer problem that fills all the buffers.
    - We can do so by adding an integer *counter* that keeps track of the number of full buffers.
    - Initially, counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer-Consumer with *counter*

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

Producer process

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

Consumer process


= modified

Race Condition

- We now show that the value of counter may be incorrect when both the producer and consumer *execute concurrently*, as follows:

`counter++` could be implemented as:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

`counter--` could be implemented as:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

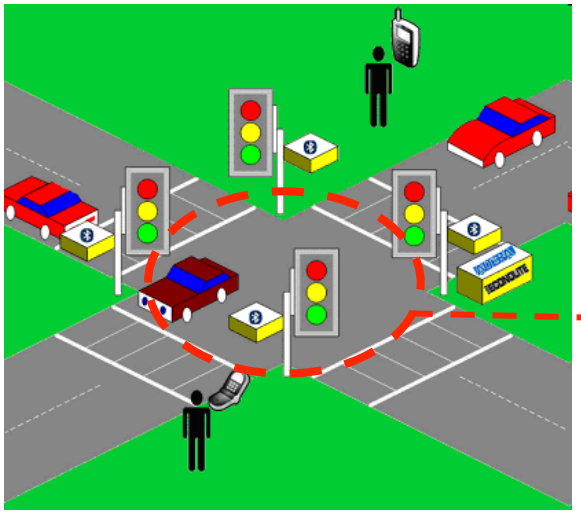
  - Consider this execution interleaving, with "counter = 5" initially:

    S0: producer execute register1 = counter {register1 = 5}
    S1: producer execute register1 = register1 + 1 {register1 = 6}
    S2: consumer execute register2 = counter {register2 = 5}
    S3: consumer execute register2 = register2 - 1 {register2 = 4}
    S4: producer execute counter = register1 {counter = 6 }
    S5: consumer execute counter = register2 {counter = 4}

- A situation like this is called a race condition,
  - 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 is taken place.

Critical Section

- The critical-section problem is to design a protocol that the processes can use to cooperate.



An analogy for a critical section

- Consider a system consisting of *n* processes $\{P_0, P_1, \ldots, P_{n-1}\}$
  - Each process has a segment of code called a *critical section*.
  - When one process is executing in its critical section, no other process is allowed to execute in its critical section.
  - Each process must request permission to enter its critical section.

General Structure of a Typical Process

```
do {
     entry section

                critical section

     exit section

                remainder section
} while (TRUE);
```

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

Solution to Critical-Section Problem

- A solution to the critical section problem must satisfy three requirements:
  1. Mutual Exclusion - If process P_i 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 there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
  3. Bounded Waiting - A bound must exist 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
     - Assume that each process executes at a nonzero speed,
     - however, no assumption concerning relative speed of the *n* processes is made

Peterson's Solution

- A classic software-based solution to the critical section problem is known as Peterson's solution.
  - This solution is not guaranteed to work on modern computers due to the way **`load`** and **`store`** perform
    - Nevertheless a good algorithmic description of a solution
  - Assuming that the **`load`** and **`store`** instructions cannot be interrupted (atomic), the Peterson's solution can be used for two processes.
- Two processes (P_0 and P_1) are required to 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 P_i is ready (i: 0 or 1)

Algorithm for Process P_i

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

               critical section

     flag[i] = FALSE;

               remainder section
} while (TRUE);
```

Note: j is equal to 1-i where i is 0 or 1

entry section

exit section

- This solution is correct, as we can show that it satisfies the three requirements (page 208 of the 9th edition of the text or page 230 of the Essential edition).

Lecture Outline

- Background
  - The Critical-Section Problem
  - Peterson's Solution
- **Synchronization Hardware**
- Semaphores
- Classic Problems of Synchronization

Synchronization Hardware

- In a uniprocessor system, the critical-section problem could be solved if we could disable interrupts (e.g. in non-preemptive kernels)
  - since current sequence of instructions would execute without preemption
  - this approach is generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

- Modern computer systems provide special hardware instructions that allow us to atomically
  - either test/modify the content of a word
  - or to swap the contents of two words
    - **Atomic** = as one non-interruptable unit

  - These instructions can be used to solve the critical-section problem in a relatively simple manner.

Synchronization Hardware (cont)

- Many systems provide hardware support for critical section code
  - We can generally state that any solution to the critical-section requires a simple tool, a lock. The *mutex lock* (***mut***ual ***ex***clusion) is the simplest of such tools.

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

- In the following we first look at a number of simple hardware instructions:
  - that are available on many systems
  - to show how they can be used effectively to solve the critical section problem.
- We abstract the main concepts behind these types of instructions by describing `TestAndSet()` and `Swap()` instructions

TestAndSet() Instruction

- The `TestAndSet()` instruction, which is executed atomically, can be defined as:

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

- If a machine supports the `TestAndSet()` instruction atomically, then we can implement mutual exclusion by declaring a Boolean variable `lock`, initialized to `false`.

```
do {
     while ( TestAndSet (&lock ))
           ; /* do nothing */

            /* critical section here */

     lock = FALSE;

            /*  remainder section here */

} while (TRUE);
```

Compare_and_Swap() Instruction

- The `Swap()` instruction, which is executed atomically, can be defined as:

```
void Compare_and_Swap (boolean *a, boolean *b, boolean *c)
{
        boolean temp = *a;
        if (*a == *b)
                *c = temp;
}
```

- Assume that each process has a local Boolean variable ***key***

Q: Do these algorithms satisfy the bounded-waiting requirement?

```
do {
   while ( Compare_and_swap(&lock,
                  FALSE, TRUE) ) {
         // Do nothing
         }
   /* critical section here */
   lock = FALSE;
   /*   remainder section   */
} while (TRUE);
```

Bounded-waiting Mutual Exclusion with `TestAndSet()`

- The following algorithm based on `TestAndSet()` satisfies all the critical-section requirements:

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;

            /* critical section here */

        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;

            /* remainder section here */

} while (TRUE);
```

Lecture Outline

- Background
  - The Critical-Section Problem
  - Peterson's Solution
- Synchronization Hardware
- **Semaphores**
- Classic Problems of Synchronization

Semaphore

- A semaphore is a less complicated and more practical synchronization tool
  - a semaphore ***S*** is an integer variable
  - ***S*** can be accessed only through two standard atomic operations:
    - `wait()` and `signal()`

```
wait (S)
{
        while (S <= 0)
            ; /* no-op */
        S--;
}
```

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

    - These two operations were originally called `P()` and `V()`

- There are two types of semaphores: **counting** and **binary**

Semaphore as General Synchronization Tool

- The value of a counting semaphore can range over an unrestricted domain
  - e.g. set to the # resources available
- The value of a binary semaphore can range only between 0 and 1;
  - binary semaphores behave like (and on most systems are also known as) mutex locks (***mut***ual ***ex***clusion),
  - and are simpler to implement
- Mutual exclusion can be implemented with binary semaphores as follows:

```
Semaphore mutex;     /*  initialized to 1  */
do {
   wait (mutex);
        /* critical Section here */
    signal (mutex);

       /* remainder section here */
} while (TRUE);
```

- Semaphores can be used to solve various synchronization problems.

Semaphore Implementation

- All modifications to the semaphore in the `wait()` and `signal()` operations must be executed indivisibly
  - i.e. when one process modifies the semaphore value, no other process can simultaneously modify the same semaphore value.

- The main disadvantage of the previous semaphore definition is that it requires busy waiting.
  - While a process is in its critical section, any other process that tries to enter its critical section must loop continuously.
    - This type of semaphore is also called a spinlock because the process spins while waiting for the lock.
  - This continuous looping is clearly a problem as it wastes CPU cycles that some other process might be able to use productively.
    - On the other hand, spinlocks do have an advantage in that no context switch is required when the process must wait on a lock,
    - so, spinlocks are useful, when locks are expected to be held for short times.

Semaphore Implementation with no Busy waiting

- To overcome the need for busy waiting, we can modify **`wait`** and **`signal`** operations.
  - Rather than engaging in busy waiting, the process can *block* itself.
- With each semaphore there is an *associated waiting queue*. Each entry has two data items:
  - a value (of type integer)
  - a pointer to next record in the list

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

- The following two operations (provided by OS as basic system calls) are used:
  - **`block`** – The ***`block()`*** operation suspends the process invoking the operation and places it on the appropriate waiting queue.
  - **`wakeup`** – The ***`wakeup()`*** operation removes one of processes in the waiting queue and place it in the ready queue (i.e. resuming the operation of a blocked process).

Implementation with no Busy waiting (cont)

- Implementation of `wait()`:

```
wait(semaphore *S)
{
   S->value--;
   if (S->value < 0) {
      /* add this process to S->list here  */
      block();
   }
}
```

- Implementation of `signal()`:

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

Problems with Semaphores

- Semaphores provide a convenient and efficient mechanism for process synchronization, though using them incorrectly can result in timing errors that are difficult to detect
  - these errors happen only if some particular execution sequence take place
- Some possible incorrect use of semaphore operations:
  - incorrect order
    - e.g. in this example, mutual exclusion is violated

```
signal (mutex)
   …
wait (mutex)
```

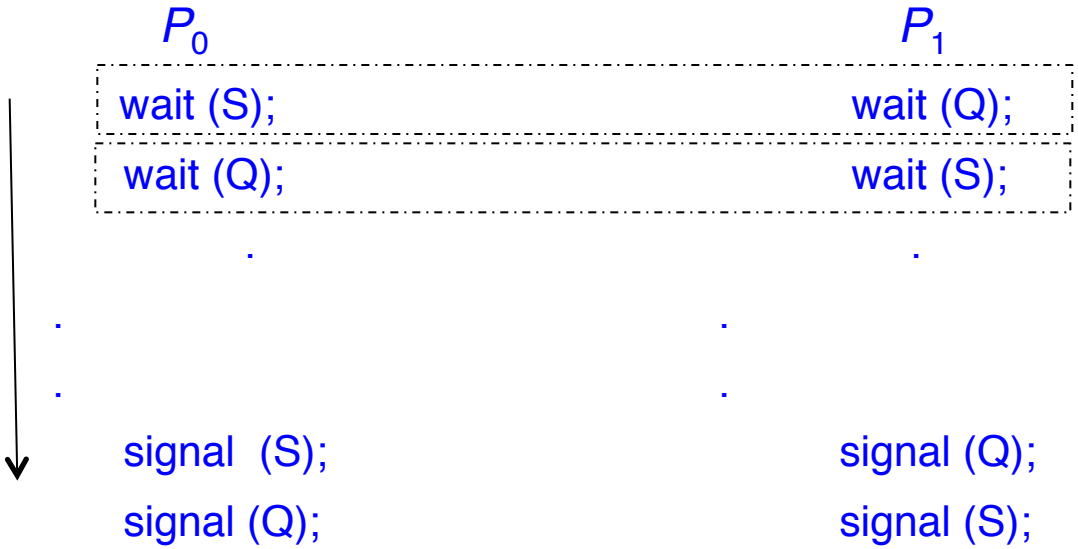
  - incorrect use
    - e.g. in this example, deadlock will occur

```
wait (mutex)
   …
wait (mutex)
```

  - Omitting `wait()` or `signal()` (or both)
    - either mutual exclusion is violated or deadlock will occur

Deadlock and Starvation

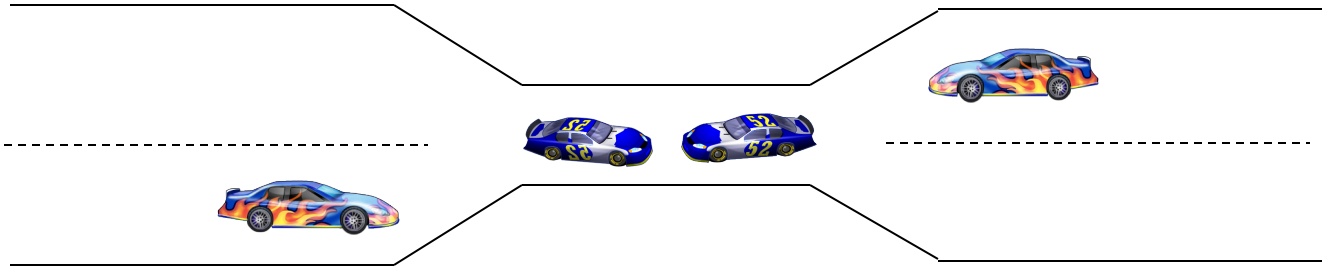
- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
  - e.g. let S and Q be two semaphores initialized to 1



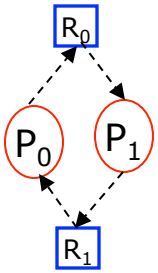
- Starvation (indefinite blocking) – A process may never be removed from the semaphore queue in which it is suspended

Deadlock (cont)

- In a deadlock, processes never finish executing, and system resources are possibly tied up, preventing other jobs from starting.



- A deadlock situation can arise if these four conditions hold simultaneously:
  - **Mutual exclusion:** only one process at a time can use a resource
  - **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
  - **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.



Critical Section Problem in Kernel Code

- At any given point of time, many kernel-mode processes may be active in the OS.
  - As a result, the code implementing an OS (kernel code) is subject to several possible race conditions.
- Two general approaches are used to handle critical sections in operating systems: depending on whether the kernel is *preemptive* or *nonpreemptive*
  - Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time (and is not preempted).
  - Preemptive kernel though must be carefully designed to ensure shared kernel data is free from race conditions. This is specially difficult to design for SMP architectures.

Q: Why, then, would we favor a preemptive kernel over a nonpreemptive one?

- a preemptive kernel is more responsive, and
- a preemptive kernel is more suitable for real-time programming

Lecture Outline

- Background
  - The Critical-Section Problem
  - Peterson's Solution
- Synchronization Hardware
- Semaphores
- **Classic Problems of Synchronization**

Classical Problems of Synchronization

- We are going to discuss a number of synchronization problems that are used for testing newly proposed synchronization schemes:
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- As discussed before, the buffer has *N* buffers, each can hold one item
- Semaphore mutex, provides mutual exclusion (initialized to the value 1)
- Semaphore full, counts the # of full buffers (initialized to the value 0)
- Semaphore empty, counts the # of empty buffers (initialized to the value N)

The structure of the producer process:

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

The structure of the consumer process:

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

Readers-Writers Problem

- Suppose a database is to be shared among a number of concurrent processes
  - *Readers* are the processes that only read the data set; they do **not** perform any updates
  - *Writers* are the processes that can both read and write
- Readers-writers synchronization problem: allows multiple readers to read at the same time. Only one single writer can access the shared data at the same time
  - There are different variants of the problem.
  - e.g. the *first* problem requires that no reader be kept waiting unless a writer already obtained permission to use the shared object. The *second* problem requires that once a writer is waiting to access the object, no new readers may start reading.
- For the solution, the processes shared the following data structures:
  - Semaphore mutex (initialized to 1)
  - Semaphore wrt (initialized to 1)
  - Integer readcount (initialized to 0)

```
Semaphore mutex, wrt;
int readcount;
```

Readers-Writers Problem (Cont.)

- The following code segment presents a solution to the readers-writers problem (which one ? first or second ?) :

The structure of a writer process:

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

The structure of a reader process:

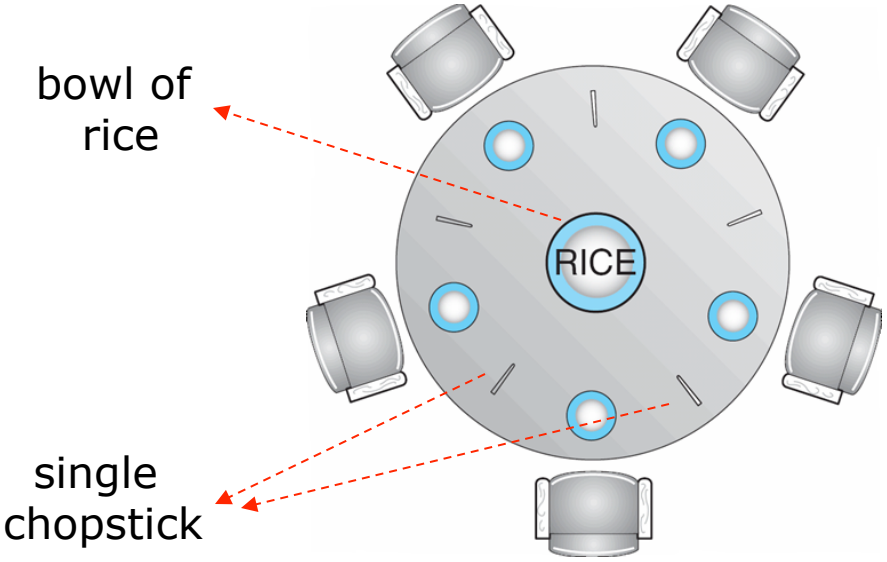
```
do {
     wait (mutex) ;
     readcount++ ;
     if (readcount == 1)
        wait (wrt) ;
     signal (mutex) ;
     /* reading is performed */
     wait (mutex) ;
     readcount-- ;
     if (readcount  == 0)
          signal (wrt) ;
     signal (mutex) ;
} while (TRUE);
```

Q: May the solution result in starvation?

- The readers-writes problem and its solutions have been generalized to provide reader-writer lock on some systems

Dining-Philosophers Problem

- The dining philosophers problem is considered a classic synchronization problem
- Consider five philosophers, who spend their lives thinking and eating:



- One simple solution starts with representing each chopstick with a semaphore
  - Semaphore chopstick [5] (initialized to 1)

Dining-Philosophers Problem: Solution

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);
```

This solution will create a deadlock. To see this, assume that all philosophers simultaneously pick up their left/right forks

Each philosopher is waiting for their left/right forks, which are being held by other philosophers who're waiting too. So they'll starve (both literally and figuratively ☺

How do you avoid starvation ?

Dining Philosophers: Solution

- The prior solution solves mutual exclusion, though it may create a deadlock. To remedy that, we may:
  - allow at most four philosophers to be sitting simultaneously, or
  - allow a philosopher to pick up her chopsticks only if both chopsticks are available, or
  - use an asymmetric solution
    - an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher does the reverse

Lecture Outline

- Background
  - The Critical-Section Problem
  - Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization