Synchronization

CS 3113

The Challenge of Concurrency

- Processes can execute concurrently
  - May be interrupted at any time, only partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

The Challenge of Concurrency

Producer-Consumer example:

- Shared circular buffer data structure:
    - Array of values: `DATATYPE buffer[BUFFER_SIZE]`
    - Number of items in the buffer: `int counter`
    - Next location to put a new item: `int in`
    - Next location to pull an item from: `int out`
- Producer and consumer processes both access these same variables in memory

Producer

```
while (true) {
     /* produce an item in next produced */

     while (counter == BUFFER_SIZE) ;
          /* do nothing */
     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Consumer

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

Possible Race Condition

- **`counter++`** could be implemented as

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

- **`counter--`** could be implemented as

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

Possible Race Condition

- Assume count = 5
- Both consumer and producer attempt to access the array at the same time
- Processes could be interleaved at the instruction level in this way:

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}

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 a table, writing a file, etc
  - When one process is in the critical section, no other may be in its critical section
- ***Critical section problem:*** design a protocol for interaction and execution that enforces non-overlapping execution of critical sections

The Critical Section Problem

***Critical section problem* -** One approach:

- Each process must ask permission to enter critical section in an **entry section** of code
- Process then executes critical section code
- Process then executes **exit section** of code
- Then, execute the **remainder section**

Critical Sections in Code

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Properties of a Proper Solution to the Critical Section Problem

1. **Mutual Exclusion:** If process $\boldsymbol{P_i}$ 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 one of these processes must be allowed to proceed
3. **Bounded Waiting:** A process that is waiting to enter its critical section can only wait for a defined amount of time

Peterson's Solution: Two Process Solution

- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- 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 P_i is ready

Algorithm for Process P_i (other Process is P_j)

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

    critical section

    flag[i] = false;
    remainder section
 } while (true);
```

Peterson's Solution

Provable that the three critical section requirements are met:

1. Mutual exclusion is preserved

   P_i enters CS only if:

   either **`flag[j] = false`** or **`turn = j`**

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Synchronization Hardware

- Many modern microprocessors provide hardware support for implementing the critical section code
- Provide mechanism that implements a **lock**
  - Then, we use the lock to protect our critical sections:
    - Must “grab” the lock before starting to execute the critical section
    - After execution, must release the lock

Synchronization Hardware

- 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 simultaneously
  - Or swap contents of two memory words

Critical Section Solution: Using A Lock

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

Test and Set Instruction

Effective behavior, but within a single instruction:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

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

Effective behavior, except it is a single instruction:

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” to the value of the passed parameter “new_value”, but only if “value” ==“expected”.

   That is, the swap takes place only under this condition.

Critical Sections with compare_and_swap()

- Shared integer “lock” initialized to 0;
- Solution:

```
do {
     while (compare_and_swap(&lock, 0, 1) != 0)
       ; /* do nothing */


    /* critical section */


 lock = 0;
    /* remainder section */
} while (true);
```

Challenges with this Use of our Hardware Solutions

Does test_and_set() satisfy our Critical Section Properties?

- Mutual exclusion: Yes
- Progress: Yes
- Bounded wait: no guarantees
  - Another process can always check the lock at the right time and capture it
  - Thus, starving another process

Bounded-waiting Mutual Exclusion with test_and_set

- lock == true -> a process is executing a critical section (or about to execute)
- lock == false -> no processes are waiting to execute a critical section
- Because we test all processes in round-robin fashion, we guarantee that each gets an opportunity to execute

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */

   // Release the lock
   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);
```

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 */

   // Release the lock
   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

Mutex Locks

- Protect a critical section by first acquire() a lock then release() the lock
  - 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(): Logical Implementation

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;
 }

release() {
    available = true;
 }
```

acquire() and release(): Usage

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

Semaphores

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore S: integer variable
  - Can only be accessed via two indivisible (atomic) operations: wait() and signal()
  - Originally called P() and V() by Dijkstra

Semaphores: Logical Definition

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

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

- Implementation guarantees safe access to S

Semaphores: Usage

- ***Binary semaphore***: integer value can range only between 0 and 1
  - Same as a mutex lock
- ***Counting semaphore***: integer value can range over an unrestricted domain
  - Can solve a wider range of synchronization problems
  - But, can still implement a Binary Semaphore

Semaphores: Usage

Consider two concurrent processes: P1 and P2

- S1 (part of P1) must happen before S2 (part of P2)
- Semaphore “synch” is initialized to 0

```
P1:
    // other code
    S1;
    signal(synch);
    // other code

P2:
    // other  code
    wait(synch);
    S2;
    // other code
```

Semaphore Details

- Implementations of `wait()` and `signal()` must guarantee that the same semaphore variable is not accessed by more than one process at the same time
- With their use, we can still have the busy waiting problem
  - Less of a problem if processes spending very little time inside of their critical sections
  - But, if processes are spending lots of time in the critical section, then busy waiting is a big problem

Semaphore Implementation with no Busy Waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer): semaphore variable
  - pointer to a FIFO queue of processes waiting on the semaphore
- Two operations:
  - **Block**: place the process invoking the operation on the appropriate waiting queue
  - **Wakeup**: remove one of processes in the waiting queue and place it in the ready queue

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

Semaphore Implementation with no Busy Waiting

Not shown: operations on the value and the queue must be atomic

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

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

Example: Bounded-Buffer Problem

- Buffer that contains n entries
- Data structure is shared by both producers and consumers
- Must protect the buffer from being accessed by more than one process at once
- Want to avoid busy-waiting in two cases:
  - Producer busy-waiting if the buffer has no room for new items
  - Consumer is busy-waiting if the buffer has no items

Example: Bounded-Buffer Problem

Data Structure:

- Semaphore **`mutex`** initialized to the value 1
  - Used to protect the buffer data structure from being accessed by more than one process
- Buffer of size ***n***
- Semaphore **`full`** initialized to the value 0
  - Counts how many items are in the buffer
- Semaphore **`empty`** initialized to the value n
  - Counts how many open spaces are in the buffer

Producer

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

Consumer

```
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);
```

Semaphores

- The version we have been working with:
  - No busy waiting. If a process wait()s on a “busy” semaphore, then it is placed into a waiting queue
  - Counting semaphores: allows us to express having some number of a specific resource type
- Producer/Consumer problem with a buffer
  - Counting semaphores to express how many used or unused slots there are in a circular buffer
  - Binary semaphore to protect the buffer data structure itself

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

Readers-Writers Solution

Shared data:

- Data set
- Semaphore **`rw_mutex`** initialized to 1
  - 1 = no readers/writers; 0 = a writer or some number of readers
- Integer **`read_count`** initialized to 0
  - Number of processes actively reading the data set
- Semaphore **`mutex`** initialized to 1
  - Protects read_count from being accessed/modified by more than one process

Writer

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

Reader

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

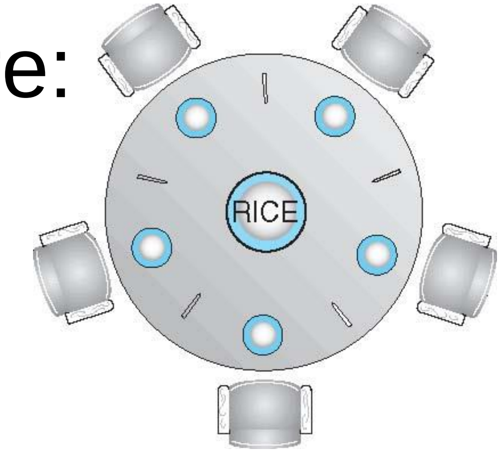
Readers-Writers Problem: Variations

- ***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
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- They don't interact with their neighbors
  - Occasionally each tries 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, the shared data are:
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem: Candidate 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);
```

What is the problem with this algorithm?

Dining-Philosophers Problem: Candidate Solution

What is the problem with this algorithm?

- We could end up with a situation where all of the philosophers have picked up exactly one chopstick
- At this stage, each is waiting for the next chopstick
- But: none will release until after another releases
- This is called ***deadlock****!*

- How do we solve this?

Dining-Philosophers Problem: A Second Solution

How do we solve the deadlock problem?

- Observation 1: at most 2 philosophers can eat at the same time (using 4 chopsticks)
- Observation 2: if we can prevent all five of the philosophers from picking up the first chopstick simultaneously, then we can guarantee that at least one can pick up the second chopstick

Dining-Philosophers Problem: A Second Solution

- Introduce another common semaphore. Call it flag
- Initialize to 4
- Before picking up the first chopstick, the philosophers must wait on the flag
- Once done with their chopsticks, they must signal the flag

Dining-Philosophers Problem: A Second Solution

The structure of Philosopher *i*:

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

                //  eat

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );
    signal (flag);
                //  think
} while (TRUE);
```

Dining-Philosophers Problem: A Second Solution

- Up to four philosophers can grab the flag at once
  - The fifth must wait until the flag becomes positive again
- This ensures that at least one philosopher can grab two chopsticks once they have the flag

Deadlock

Deadlock: two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Starvation: Indefinite Blocking

A process may never be removed from the semaphore queue in which it is suspended

- The semaphore/mutex might still be released, but another waiting process can get it first

Problems with Semaphores

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

Next Topic: Deadlock

- Formal definition
- Techniques for preventing it