CSE 4/521
Introduction to Operating Systems

Lecture 6 – Process Synchronization I

(Background, The Critical-Section Problem, Synchronization Hardware, Mutex Locks, Semaphores)

Summer 2018

Overview

Objective:

1. To present the concept of process synchronization
2. To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
3. To present both software and hardware solutions of the critical-section problem

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Mutex Locks
- Semaphores

Recap

- Overview
  - Process creation vs. Thread creation, benefits of threads
- Multicore Programming
  - Difference between Parallelism and Concurrency
- Multithreaded Models
  - 3 types: Many-to-one, one-to-one, many-to-many
- Thread Libraries
  - Pthreads and Java threads
- Implicit Threading
  - Threads created and management by Thread pools and OpenMP
- Operating-System Examples
  - Windows threads, Linux threads

Questions

1. Is is possible to have concurrency but not parallelism? (Easy)
2. What is the most prevalent threading model currently? Why is it better than others? (Medium)
3. How many unique processes and threads are created? (Hard)

```
pid_t pid;
pid = fork();
if (pid == 0) {
        fork();
        thread_create(. . .);
}
fork();
```

Overview

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Mutex Locks
- Semaphores

Overview

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Mutex Locks
- Semaphores

Background

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

Background

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 */
}
```

counter++ implemented as

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

counter-- implemented as

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

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

S0: producer execute `register1 = counter` {register1 = 5}

Background

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 */
}
```

---

counter++ implemented as

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

counter-- 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}

Background

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 */
}
```

counter++ implemented as

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

counter-- 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}

Background

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 */
}
```

---

counter++ implemented as

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

counter-- 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}

Background

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 */
}
```

---

counter++ implemented as

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

counter-- 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 }

Background

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 */
}
```

---

counter++ implemented as

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

counter-- 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}

Background

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_consume
}
```

counter++ implemented as

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

counter-- implemented as

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

Consider this execution interleavin ... itially:

S0: producer execute regis ... {register1 = 5}

S1: producer execute r ... ister1 + 1 {register1 = 6}

S2: consumer exe ... counter {register2 = 5}

S3: consum ... r2 = register2 – 1 {register2 = 4}

S4: pr ... nter = register1 {counter = 6 }

... e counter = register2 {counter = 4}

INCONSISTENT COUNTER VALUE BETWEEN PRODUCER AND CONSUMER

(RACE CONDITION)

Overview

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Mutex Locks
- Semaphores

Critical Section Problem

- 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
- Each process must ask permission to enter critical section in entry section, after critical section comes exit section, then remainder section

```
do {
    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Critical Section Problem – 3 Important Guarantees to be provided

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

Critical Section Problem – Critical Section Handling in OS

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

Overview

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Mutex Locks
- Semaphores

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of locking
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - OSes using this not scalable to multiprocessor systems
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptible
  - Either test memory word and set value – `test_and_set()`
  - Or swap contents of two memory words – `compare_and_swap()`

Synchronization Hardware – Idea of Locking

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

Synchronization Hardware – `test_and_set()` Instruction

Definition:

```
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".

Solution:

shared boolean variable `lock`, initialized to FALSE

```
do {
          while (test_and_set(&lock))
             ; /* do nothing */
                 /* critical section */
          lock = false;
                 /* remainder section */
       } while (true);
```

Synchronization Hardware – `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;
     }
```

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

Solution:

Shared integer `lock` initialized to 0;

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

Synchronization Hardware – 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);
```

Initialization for all Processes $\mathbf{P}_i$:

```
Boolean waiting[n] = false;
Boolean lock = false
```

Overview

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Mutex Locks
- Semaphores

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- Simplest OS-level approach called 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

Mutex Locks

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

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

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

Overview

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Mutex Locks
- Semaphores

Semaphores

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process 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()`**
- Definition of the `wait()` operation

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

- Definition of the `signal()` operation

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

Semaphores - Usage

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1
  - Same as a mutex lock
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore synch initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphores - Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- In semaphores, the `wait()` and `signal()` code gets placed in the critical section
  - Could now have busy waiting in critical section implementation
  - Little busy waiting if critical section rarely occupied

Semaphores – Implementation with no Busy-waiting

- With each semaphore there is an associated waiting queue
- 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;
```

Semaphores – Implementation with no Busy-waiting

```
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);
   }
}
```

Next Lecture

- Semaphores cont. – (Deadlocks and Starvation)
- Classic Problems of Synchronization
- Synchronization Examples

Credits for slides

Silberschatz, Galvin and Gagne