CS 1550: Introduction to Operating Systems

Recitation 4: InterProcess Communication (IPC)

Mohammad Hasanzadeh Mofrad

University of Pittsburgh

moh18@pitt.edu

9/16/2019

Process Synchronization

- **Inter-Process Communication (IPC)** refers to the mechanisms that allows processes to work on shared data e.g. client and server architecture where client requests data and the server responds to client requests.
- Motivations
  - Cooperating processes share a logical address space
  - Avoiding inconsistency while having many concurrent data access
  - In massively parallel algorithms, a process may be interrupted at any point
  - Enable multithreading while having multiple cores

Producer/consumer example

- Producer – consumer problem (bounded-buffer problem) is a classic example of multi-process synchronization problem which consists of:
  - A fixed-size buffer used as a queue.
  - A producer which generates data and put it into the buffer.
  - A consumer which consumes data and remove it from the buffer

https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem

Producer/consumer example

```
/* Producer */
/* produce an it
next_produced */
while (true)
{
      while (counter == BUFFER SIZE)
            ;  /* do nothing */
      buffer[in] = next_produced;
      in = (in + 1) % BUF
      counter++;
}
```

R_1 = counter
$R_1 = R_1 + 1$
counter = R_1

Counter = 5

Counter = {4, 5, or 6}

```
/* Consumer */
/* consume the it
next_consumed */
while (true)
{
      while (counter == 0)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER SIZE;
      counter--;
}
```

R_2 = counter
$R_2 = R_2 – 1$
counter = R_2

Example (counter = 5)

/* Producer */

R_1 = counter (R_1 = 5)

R_1 = R_1 + 1 (R_1 = 6)

Φ

Φ

counter = R_1 (counter = 6)

Φ

/* Consumer */

Φ

Φ

R_2 = counter (R_2 = 5)

R_2 = R_2 – 1 (R_2 = 4)

Φ

counter = R_2 (counter = 4)

counter = 4

Example (counter = 5)

```
/* Producer */
R1 = counter (R1 = 5)
R1 = R1 + 1     (R1 = 6)
Φ
Φ
Φ
counter = R1 (counter = 6)
counter = 6
```

```
/* Consumer */
Φ
Φ
R2 = counter (R2 = 5)
R2 = R2 – 1     (R2 = 4)
counter = R2 (counter = 4)
Φ
```

Example (counter = 5)

```
/* Producer */
R₁ = counter (R₁ = 5)
R₁ = R₁ + 1      (R₁ = 6)
counter = R₁ (counter = 6)
Φ
Φ
Φ
```

```
/* Consumer */
Φ
Φ
Φ
R₂ = counter (R₂ = 6)
R₂ = R₂ - 1      (R₂ = 5)
counter = R₂ (counter = 5)
counter = 5
```

Critical Section Problem

```
/* Typical control flow for a
process pi */
do
{
     entry section
          critical section
     exit section
          non critical section
}
while (TRUE)
```

- Concurrent accesses to a shared resource can lead to an unexpected behavior. So, the part of the program which is accessed concurrently is protected. This protected section is called **Critical Section**.
- N processes: P = {p_0, p_1, …, p_n}
  - Change a variable
  - Update a table
  - Write to a file

Solution Requirements for Critical Section

- **Mutual Exclusion**
  - Only p_i can be in it's critical section at a given point of time
- **Progress**
  - If the critical section is empty, processes must be able to enter it
- **Bounded waiting**
  - No process should wait forever to enter it's critical region
- How OS handles critical section
  - Non-preemtive kernels
    - Free from race conditions
    - Starvation problem
  - Preemptive kernels
    - Difficult to design
    - More responsive
- Hardware based solutions

Peterson's Solution

```
/* Process pi in Peterson's solution*/
int turn;
boolean flag[2];
do {
     flag[i] = true
     turn = j
     while (flag[j] && turn == j)
          ; /* do nothing */
     critical section
     flag[i] = false;
     non critical section
}
while (TRUE)
```

- A classic software-based solution which is working
- Restricted to 2 processes working on their critical sections
  - **j = 1 - i**
- **turn** indicates whose turn is it?
- **flag** indicates a process is ready to enter it's critical region or not

Proving CS Requirements for PS

1. Mutual Exclusion

2. Bounded Waiting

3. Progress

```
/* P0 */
do {
      flag[0] = true
      turn = 1
      while (flag[1] && turn == 1)
            ; /* do nothing */
      critical section
      flag[0] = false;
      non critical section
}
while (TRUE)
```

```
/* P1 */
do {
      flag[1] = true
      turn = 0
      while (flag[0] && turn == 0)
            ; /* do nothing */
      critical section
      flag[1] = false;
      non critical section
}
while (TRUE)
```

MUTEX (MUTual EXclusion)

- **Race condition**: A race condition is an undesirable condition that happened when having multiple processes running on a piece of data which does not use any exclusive locks to control access.
- A MUTEX is a LOCK for CRITICAL SECTION and thus prevents RACE CONDITION
- Mutexes are the simplest synchronization tools in the operating system

MUTEX

```
/* Typical control flow for mutex
locks */
do
{
      acquire() lock
            critical section
      release() lock
            non critical section
}
while (TRUE)
```

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

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

MUTEX

+ Atomicity: Implemented using hardware mechanisms

    test and set() instruction

    compare and swap() instruction

- Busy waiting: process spins while waiting for the lock spinlock

SEMAPHORE

```
/* semaphore(s) */
up(s)
{
      s++;
}

down(s)
{
      while (s <= 0)
            ; /* busy wait */
      s--;
}
```

- A classic software-based solution which is working
- A more sophisticated mutex lock
- All modifications to semaphore must be indivisible
- No 2 processes can modify a semaphore simultaneously
- Counting / binary semaphores

Semaphore with blocking

```
class Semaphore {
    int value;
    ProcessList pl;

    void down () {
        value -= 1;
        if (value < 0) {
            // add this process to pl
            pl.enqueue(currentProcess);
            Sleep();
        }
    }

    void up () {
        Process P;
        value += 1;
        if (value <= 0) {
            // remove a process P from pl
            P = pl.dequeue();
            Wakeup(P);
        }
    }
}
```

Producer/Consumer with semaphores

```
const int n;
Semaphore empty(n),full(0),mutex(1);
Item buffer[n];
```

```
Producer
int in = 0;
Item pitem;
while (1) {
  // produce an item
  // into pitem
  empty.down();
  mutex.down();
  buffer[in] = pitem;
  in = (in+1) % n;
  mutex.up();
  full.up();
}
```

```
Consumer
int out = 0;
Item citem;
while (1) {
  full.down();
  mutex.down();
  citem = buffer[out];
  out = (out+1) % n;
  mutex.up();
  empty.up();
  // consume item from
  // citem
}
```