Process Synchronization

Hui Chen [a]

[a]CUNY Brooklyn College

October 30, 2024

Outline

Outline

Race Condition

Without orderly executation of cooperating processes (or threads), concurrent access to shared data may result in data inconsistency, called *race condition*.

Producer-Consumer Problem

- Let's consider the following solution to the *Producer-Consumer* problem where we use a counter to track buffer use.

Shared Buffer

```
1 #define BUFFER_SIZE 10
2 typedef struct { } item;
3
4 // The following are shared among cooperating processes
5 item buffer[BUFFER_SIZE];
6 int in = 0;
7 int out = 0;
8 int counter = 0;
```

Producer

```
1 while (true) {
2   /* produce an item in next produced */
3   while (counter == BUFFER_SIZE)
4     ; /* do nothing */
5   buffer[in] = next_produced;
6   in = (in + 1) % BUFFER_SIZE;
7   counter ++;
8 }
```

Consumer

```
1 while (true) {
2   while (counter == 0)
3     ; /* do nothing */
4   next_consumed = buffer[out];
5   out = (out + 1) % BUFFER_SIZE;
6   counter--;
7   /* consume the item in next consumed
        */
8 }
```

Counter and Machine Code

- Assume the compiler generates the machine code whose pseudo-code is as follows,
- `counter ++`

```
1       register1 = counter
2       register1 = register1 + 1
3       counter = register1
```

- `counter -`

```
1       register2 = counter
2       register2 = register2 - 1
3       counter = register2
```

Program Execution Scenario

- Context switches happen and result in the following sequence of execution

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

Getting Available Process ID

Let's consider the design of the `fork()` system call in the OS kernel.

- The OS kernel creates child processs using the fork() system call and assigns the process a unique process ID.

```
1 int fork() {
2   ...
3   pcb.pid = get_next_available_pid();
4   ...
5 }
```

- Consider two processes P_1 and P_2 are calling `fork()` to create two child processes.
- Can the OS kernel assign the same `pid` to the two child processes (without proper synchronization)?

More Examples

Let's do a couple of experiments and observer race conditions...

1. Incrementing an integer in multiple Java threads
2. Simulating `next_available_pid()`
3. Simulating the producer-consumer problem using shared memory (without proper synchronization)
4. Reading and writing to global variables in two threads.

Incrementing an integer in multiple Java threads

If you choose to download, compile, and run it in the Linux system, follow these steps

```
1 sudo apt-get install -y default-jdk
2 mkdir IncrementInt
3 cd IncrementInt
4 wget https://raw.githubusercontent.com/huichen-
    cs/OSClassExamples/master/synchronization/
    racecond/incrementint/IncrementInt.java
5 javac IncrementInt.java
6 java IncrementInt
```

Outline

Critical Section Problem

- Consider system of n processes $\{P_0, P_1, \ldots, P_{n-1}\}$
- Each process has critical section segment of code where the 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
- The 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

General Structure of Processes with Critical Section

```
1 do {
2    // entry section
3    ...
4    // critical section
5    ...
6    // exit section
7    ...
8    // remainder section
9    ...
10 } while (true);
```

Assumptions and Requirement

- Assumptions
  1. Assume that processes execute at a nonzero speed.
  2. There is no assumption concerning relative speed of the processes.
- 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.

Outline

- Generally speaking, any solution to the critical-section problem is to construct a simple tool, called a “lock“
- A process must acquire a lock before entering a critical section, and releases the lock when it exits the critical section

Solution to Critical-section Problem Using Locks

- Let's compare the following two pseudo-code snippets,

General structure of a process with critical section

```
1  do {
2    // entry section
3    ...
4    // critical section
5    ...
6    // exit section
7    ...
8    // remainder section
9    ...
10 } while (true);
```

General solution to Critical Section problem using locks

```
1  do {
2    // acquire lock
3    ...
4    // critical section
5    ...
6    // release lock
7    ...
8    // remainder section
9    ...
10 } while (true);
```

Mutex Locks and Semaphores

- Mutex locks.
  - Using a Boolean variable to indicate if the lock is available or not.
  - Defining two operations, `acquire()` and `release()` to acquire and release the lock
  - `acquire()` and `release()` must be atomic (indivisible)
- Semaphores.
  - Using an integer variable indicates if the lock is available or not.
  - Defining two operations, `wait()` (or `P()`, or `down()`) and `signal()` (or `V()` or `up()`) to acquire and release the lock
  - `wait()` (or `P()`) must be atomic.
  - Counting semaphore. The integer value can range over an unrestricted domain.
  - Binary semaphore. The integer value can range only between 0 and 1, essentially, a Mutex lock.

Mutual Exclusion via Mutex Locks

Let's increment a shared variable in multiple processes/threads ...

Mutual Exclusion via Semaphores

Let's increment a shared variable in multiple processes/threads ...

Control Execution Using Semaphores

Let's consider P_1 and P_2 that require S_1 in P_1 to happen before S_2 in P_2 ...

1. Create a semaphore "synch" initialized to 0
2. Implement P_1 as follows,

```
1    S1;
2    signal(synch);
```

3. Implement P_2 as follows,

```
1    wait(synch);
2    S2;
```

Question. Can we realize the above using a Multex lock instead?

Implementation of Mutex Locks

- Mutex locks. An implementation of `acquire()` and `release()` is via hardware atomic instructions such as `compare-and-swap` and `test-and-set`.
- This implementation of Mutex locks requires busy waiting. We call a Mutex lock whose implementation requires busy-waiting a *spinlock*.

```
1 acquire() {
2    while (!available)
3       ; /* busy wait */
4    available = false;
5 }
6
7 release() {
8    available = true;
9 }
```

Implementation of Semaphores

Let's consider an implementation without busy waiting ...

```
1  // S->list is a list of processes that are in the sleeping state
2  wait(semaphore *S) {
3      S->value--;
4      if (S->value < 0) {
5          // add this process to S->list;
6          S->list.add(this_process);
7          block();
8      }
9  }
10
11 signal(semaphore *S) {
12     S->value++;
13     if (S->value <= 0) {
14         // remove a process P from S->list;
15         S->list.remove(P);
16         wakeup(P);
17     }
18 }
```

Outline

Monitor

- It is easy to make mistakes when using semaphores.
- To reduce such mistakes, introduce *Monitor*.
  - A high-level abstraction that provides a convenient and effective mechanism for process synchronization
  - An Abstract Data Type, internal variables only accessible by code within the procedure
  - Only one process may be active within the monitor at a time

Condition Variable

Conditional variables are introduced to address several issues:

- to avoid busy-waiting using synchronization mechanism like mutex locks,
- to use/share a single lock for a common, but subtly different synchronization needs.

Two operations are allowed on a condition variable:

- `x.wait()`. A process that invokes the operation is suspended until `x.signal()`
- `x.signal()`. Resumes one of processes (if any) that invoked `x.wait()`; however, if no x.wait() on the variable, then it has no effect on the variable

Further Study

- Example programs
- OS examples
- Implementations
- ...

Outline

Events vs. Threads

There have been a recurrent discussion on how we should realize concurrency [2, 3, 4, 6, 5]

- Threads vs. events [2, 4, 6, 5]
- Theory vs. practice ([1, Section 9.1], [3])

Reference I

[1] Brian Goetz et al. *Java concurrency in practice*. Pearson Education, 2006.

[2] Hugh C Lauer and Roger M Needham. "On the duality of operating system structures". In: *ACM SIGOPS Operating Systems Review* 13.2 (1979), pp. 3–19.

[3] John Ousterhout. "Why threads are a bad idea (for most purposes)". In: *Presentation given at the 1996 Usenix Annual Technical Conference*. Vol. 5. San Diego, CA, USA. 1996.

[4] J Robert Von Behren, Jeremy Condit, and Eric A Brewer. "Why Events Are a Bad Idea (for High-Concurrency Servers).". In: *HotOS*. 2003, pp. 19–24.

Reference II

[5] Rob Von Behren et al. "Capriccio: scalable threads for internet services". In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 268–281.

[6] Matt Welsh, David Culler, and Eric Brewer. "SEDA: an architecture for well-conditioned, scalable internet services". In: *ACM SIGOPS Operating Systems Review* 35.5 (2001), pp. 230–243.