Operating System(Server)

Lecture 4
Process Synchronisation 1

Dr. Kevin Farrell

Chapter 7 Part 1: Process Synchronization

- Background
  - The Bound-Buffer Produce-Consumer Problem revisted
  - Race Conditions
- The Critical-Section Problem, and attempts at its solution:
  - Two-Process Solution: Strict Alternation
  - Two-Process Solution: un-named algorithm
  - Two-Process Solution: Peterson's Solution
  - Multiple-Process Solution: Bakery Algorithm
- Synchronization Hardware
  - `TestAndSet` Instruction
  - `Swap` Instruction

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Recall the shared-memory solution to bounded-butter Producer-Consumer problem (Lecture/Chapter 4)
- For readability, let *N represent the quantity BUFFER_SIZE, for the size of the buffer*
- Our earlier solution allows at most *N – 1* items in buffer at the same time. A solution, where all *N* items in the buffer can be used is not simple. Let's try to find one:
  - Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer

Bounded-Buffer... New Solution?

- Shared data: all shared data as before, and:
  - Introduce new ***counter*** **variable**

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer

- Producer process: changed code

```
item nextProduced;

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

Bounded-Buffer

- Consumer process: changed code

```
item nextConsumed;

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

Bounded Buffer

- The statements

  counter++;
  counter--;

  must be performed *atomically* in order to guarantee a consistent value for **counter**

- Atomic operation means an operation that completes in its entirety without interruption.

Bounded Buffer

- The statement "**counter++**" may be implemented in assembly language as:

  register1 = counter
  register1 = register1 + 1
  counter = register1

- The statement "**counter- -**" may be implemented as:

  register2 = counter
  register2 = register2 – 1
  counter = register2

- It is clear that **counter++** and **counter- -** are NOT performed atomically, since they consist of THREE distinct machine language statements, which could be scheduled separately by the OS.

Bounded Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- This will result in inconsistent values!

- Interleaving depends upon how the producer and consumer processes are scheduled by the OS.

Bounded Buffer

Example of interleaving which results in inconsistent values:

- Assume **counter** is initially 5. One possible interleaving of the assembly language statements is:

  producer: **register1 = counter** (*register1 = 5*)
  producer: **register1 = register1 + 1** (*register1 = 6*)
  consumer: **register2 = counter** (*register2 = 5*)
  consumer: **register2 = register2 – 1** (*register2 = 4*)
  producer: **counter = register1** (*counter = 6*)
  consumer: **counter = register2** (*counter = 4*)

- Here, final value of **counter** = 4

Bounded Buffer

Example of interleaving which results in inconsistent values:

- Assume **counter** is initially 5. *Another* interleaving of statements is:

  producer: **register1 = counter** (*register1 = 5*)
  producer: **register1 = register1 + 1** (*register1 = 6*)
  consumer: **register2 = counter** (*register2 = 5*)
  consumer: **register2 = register2 – 1** (*register2 = 4*)
  consumer: **counter = register2** (*counter = 4*)
  producer: **counter = register1** (*counter = 6*)

- Here, final value of **counter** = 6
- The value of **counter** may be either 4 or 6 depending on whether the consumer or producer finishes last!
- However, the correct result should be 5

Race Condition

- **Race condition**: The situation where several processes access --- and manipulate --- shared data concurrently. The final value of the shared data depends upon which process finishes last.

- In our Producer-Consumer problem, we need to ensure that only one process at a time can be manipulating the variable *counter*. i.e. the Producer and Consumer processes need to be synchronised!

- To prevent race conditions, concurrent processes must be **synchronised**.

The Critical-Section Problem

- *Consider n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem: ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section:
  - In other words, we say that the execution of critical sections by the processes must be *mutually exclusive in time*
- *The critical-section problem* is to design a protocol that the processes can use to cooperate
  - Each process must request permission to enter its critical section.
  - The section of code implementing this request is the **entry section**
  - The critical section may be followed by an **exit section**
  - The remaining code is the **remainder section**

Solution to Critical-Section Problem: 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. (No process running outside its critical section may block other processes).
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. (No process should have to wait forever to enter its critical section).
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the *n* processes.

Initial Attempts to Solve Critical Section Problem

In the next number of slides, we will work up to solutions to the critical-section problem that satisfy the three requirements. For that purpose, we consider:

- Only 2 processes, P_i and P_j
- General structure of process P_i (other process P_j)

do {

entry section

critical section

exit section

reminder section

} **while (1)**;

- Processes may share some common variables to synchronize their actions.

Two-Process Solutions: Algorithm 1: Strict Alternation

- Two Processes, P_0 and P_1 $\Rightarrow$ $j == 1 - i$
- Shared variables used for synchronisation purposes:
  - **int turn**;
    initially **turn = 0**
  - **If turn == i** $\Rightarrow$ P_i can enter its critical section
- Process P_i

```
do {
    while (turn != i)
        ; // Busy wait
    critical section
    turn = 1 - i;
    reminder section
} while (1);
```

- Satisfies mutual exclusion condition, but not progress condition.

Algorithm 1: Strict Alternation fails

- Satisfies mutual exclusion, but not progress. Explanation:
- Initially,the integer variable, ***turn*** = 0, keeps track of whose turn it is to enter the critical section (CS), and update the shared memory.
- Initially, P_0 inspects turn, finds it to be 0 and enters its CS.
- P_1 also finds it to be 0 and begins a waiting loop which continually tests the value of turn. This is called *Busy Waiting*. It should be avoided if wait time is long as it wastes CPU time.
- When P_0 leaves CS, it sets turn = 1 and allows P_1 to enter its CS
- Now suppose P_1 finishes its CS so both processes are in non-CS with turn = 0.
- Now P_0 executes its **while** loop quickly, entering and then exiting its CS, and setting turn=1. Both processes are then executing in their non-CS regions.
- P_0 then finishes its non-CS and re-loops. But it is not permitted to enter its CS now since **turn** = 1 so it loops waiting for P_1 to set **turn** = 0.
- However, this violates condition 2 (Progress) where a process is being blocked by a process NOT in its critical section.

Two-Process Solutions: Algorithm 2

- Shared variables
  - **boolean interested[2]**;
    initially **interested [0] = interested [1] = false.**
  - **interested [i] = true** ⇒ P_i ready to enter its critical section
- Process P_i

```
do {
        interested[ i ] = true;
        while (interested[ 1 - i ]==TRUE)
                ; // Busy wait
        critical section
        interested[ i ] = false;
        remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

Two-Process Solutions: Algorithm 3: Peterson's Solution

- Combined shared variables of algorithms 1 and 2.
- Existing Shared variables
  - **turn**
  - **boolean interested[2]**
- Process P_i

```
do {
    interested[ i ] = true;
    turn = 1 - i;
    while (  (interested[ 1 - i ]==TRUE) AND (turn == 1 - i) )
              ; // Busy wait
    critical section
    interested[ i ] = false;
    remainder section
} while (1);
```

- <u>Satisfies all three requirements</u> ⇒ <u>solves the critical-section problem for two processes</u>.

Multiple-Process Solutions: Bakery Algorithm

Critical section for *n* processes

- Before entering its critical section, process receives a number (ticket). Holder of the smallest number enters the critical section.
- The subscript on each process, indicates the order in which the process entered the operating system (not the critical section); i.e. creation time.
- In the case of processes P_i and P_j receiving the same number, if $i < j$, then P_i is served first (since it existed in the OS before P_j), otherwise P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Bakery Algorithm

- Notation <≡> lexicographical order (ticket #, process id #)
  - Definition: $(a,b) < (c,d)$ if $a < c$ or if ($a == c$ **and** $b < d$)
  - Definition: max $(a_0,\ldots, a_{n-1})$ is a number, k, such that $k \geq a_i$ for $i = 0, \ldots, n - 1$
- Shared data
  - **boolean choosing[n];**
  - **int number[n];**
- Data structures are initialized as follows:
  - **choosing[n] = false, for all n**
  - n**umber[n] = 0, for all n**

Bakery Algorithm

```
do {
    choosing[ i ] = true;
    number[ i ] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[ i ] = false;
    for (j = 0; j < n; j++) {
      while (choosing[ j ] == TRUE)
            ; // Busy wait
      while ((number[ j ] != 0) && ( (number[ j ], j ) < (number[ i ], i )  )  )
            ; // Busy wait
    }
    critical section
    number[ i ] = 0;
    remainder section
} while (1);
```

Bakery Algorithm

- ***To prove the (correctness of the solution for the) bakery algorithm, we need first to show that the 2nd while statement is true when j = i. i.e. we wish to show that if,***
  - ***P_i is in its critical section, and***
  - ***P_k (k != i) has already chosen its number[k] != 0***
- ***then,(number[i], i) < (number[k], k)***
- ***Prove it!!!***
- ***Given this result, it is simple to show that mutual exclusion holds:***
  - ***Consider P_i in its CS, and P_k trying to enter the P_k CS***
  - ***When process, P_k executes the 2nd while statement for j == i, it finds that:***
    - ✓ ***number[i] != 0***
    - ✓ ***(number[i], i) < (number[k], k)***
  - ***It thus continues looping in the while statement until P_i leaves the P_i CS.***
  - ***Progress and Bounded-waiting conditions are met because processes enter their CS on a first-come, first-served basis***

Synchronization Hardware

- Often, hardware features make the task easier of having to program “synchronisation code”
- Hardware features also generally improve system efficiency as they are faster
- In the next number of slides we present some simple hardware instructions which are available on many systems, and show how some of them can be used effectively in solving the critical-section problem.

Synchronization Hardware: Disabling Interrupts

- In *uniprocessor* systems, we could solve the critical-section problem, simply by **disabling** (forbidding) interrupts to occur while a shared variable is being modified
  - When a process enters its CS, interrupts would be disabled, and re-enabled after it left its CS
  - therefore a process could not be interrupted while executing (it could not be pre-empted)
- Useful technique for OS kernel, but not appropriate as a general mutual exclusion mechanism for user processes
- In the case of *multiprocessor* systems, it should generally **not** be used, since:
  - Disabling interrupts is time-consuming for the message to be passed to all processors
  - Message-passing delays entry into each CS, and system efficiency decreases

Synchronization Hardware: The `TestAndSet` Instruction

- Test and modify the content of a word *atomically*
  .

```
boolean TestAndSet(boolean &target) {
	boolean rv = target;
	target = true;

	return rv;
}
```

Mutual Exclusion with Test-and-Set Lock TSL Instruction

- Shared data:

```
boolean lock = false;
```

- Process P_i

```
do {
    while (TestAndSet(&lock) == true)
        ; // Do nothing
    critical section
    lock = false;
    remainder section
} while(1);
```

Synchronization Hardware: `Swap` Instruction

- *Atomically* swap two variables.

```
void Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```

Mutual Exclusion with Swap

- Shared data:
```
boolean lock = false; // initialised to false
boolean key;
```
- Process P_i
```
do {
  key = true;
  while (key == true)
            Swap(lock,key);
  critical section
  lock = false;
  remainder section
} while(1);
```

- **Neither the `TestAndSet` instruction nor the `Swap` instruction satisfy the bound-waiting requirement**