BUCKNELL UNIVERSITY
Computer Science

CSCI 315 Operating Systems Design

Process Synchronization

Notice: This set of slides is based on the notes by Professor Perrone of Bucknell and the textbook authors Silberschatz, Galvin, and Gagne

Two Examples

- Multiple threads increment a shared variable leading to incorrect results
  - http://www.eg.bucknell.edu/~cs315/Fall13/code/thread/trd-share.c
- Multiple threads share a string buffer (read/write) leading to incorrect results
  - http://www.eg.bucknell.edu/~cs315/Fall13/code/synch/consumer-producer-wosynch.c

Process Synchronization

- Processes work together to solve problems
- They need to collaborate with each other in order to accomplish a task
- Without collaboration, things can go wrong

Race Condition

A **race condition** is where the outcome of the execution depends on the particular order in which the threads[note] access the shared data.

Note: in this context, we will use the term process and thread interchangeably.

We have seen this phenomenon in our thread discussion

```
[xmeng@polaris thread]$ ./trd-share
main() reporting that all 5000 threads have terminated
v should be 5000, it is 4998
[xmeng@polaris thread]$
```


The Synchronization Problem

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

Producer-Consumer
Race Condition

The **Producer** does:

```
while (1) {
    while (count == BUFFER_SIZE)
        ; // do nothing
    // produce an item and put in nextProduced
    buffer[in] = make_item();
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Producer-Consumer
Race Condition

The **Consumer** does:

```
while (1) {
    while (count == 0)
        ; // do nothing
    item =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    // consume the item
}
```


Producer-Consumer
Race Condition

- count++ could be implemented as

```
lw $t0, 0($s0)    # load memory content at $s0 to $t0
addi $t0, $0, 1   # increment $t0 by 1
sw $t0, 0($s0)    # store content in $t0 to memory at $s0
```

- count-- could be implemented as

```
lw $t1, 0($s0)    # load memory content at $s0 to $t1
subi $t1, $0, 1   # decrement $t1 by 1
sw $t1, 0($s0)    # store content in $t1 to memory at $s0
```

- Consider this execution interleaving when count == 5:

```
Step 0: producer execute lw $t0, 0($s0)   # $t0 == 5
Step 1: producer execute addi $t0, $0, 1  # $t0 == 6
Step 2: consumer execute lw $t1, 0($s0)   # $t1 == 5
Step 3: consumer execute subi $t1, $0, 1  # $t1 == 4
Step 4: producer execute sw $t0, 0($s0)   # count == 6
Step 5: consumer execute sw $t1, 0($s0) # count == 4
```


The Critical-Section Problem

- It turns out that the consumer-producer problem is one particular problem in a general category of problems called *the critical-section problem:*
  - A collection of collaborating processes, each of which has a segment of code (*critical section*) that accesses some common data. To ensure the correctness of the result, only one process can enter its critical section to access the shared data at any time.
  - The *critical-section problem* is to design a protocol that ensures the correctness of the result under such a condition.

The Critical-Section Problem
Solution 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. (Assume that each process executes at a nonzero speed. No assumption concerning relative speed of the **N** processes.)

Typical Process P_i

```
do {
  entry section
     critical section
  exit section
     remainder section
} while (TRUE);
```


OpenMP Code Example

```
#include <omp.h>
#include <stdio.h>
int main(int argc, char *argv[])  {
    /* sequential code */
    int v = 0;

    #pragma omp parallel shared(v)
    {
        #pragma omp critical (addv)
        {
            v ++;
        }
        printf("I am a parallel region\n");
    }
    /* sequential code */
    printf("value of v = %d\n", v);
    return 0;
}
```

How To Synchronize Processes?

- OpenMP provides a nice solution for programmers.
- But how are they implemented? How do we approach a synchronization problem in general?
- There could be hardware solution to this problem as well. We are concentrating on software solutions for now.

Peterson's Solution
for a 2-process case


```
int turn;
boolean flag[2];
```

```
do {
  flag[i] = TRUE;
  turn = j;
  while (flag[j] && turn == j);
     critical section
  flag[i] = FALSE;
     remainder section
} while (TRUE);
```


Peterson's Solution
Process 0


```
int turn;
boolean flag[2];
```

```
do {
  flag[0] = TRUE;
  turn = 1;
  while (flag[1] && turn == 1);
     critical section
  flag[0] = FALSE;
     remainder section
} while (TRUE);
```


Peterson's Solution
Process 1


```
int turn;
boolean flag[2];
```

```
do {
  flag[1] = TRUE;
  turn = 0;
  while (flag[0] && turn == 0);
     critical section
  flag[1] = FALSE;
     remainder section
} while (TRUE);
```


Limitation to Peterson's Solution

- Strict order of execution
- Variable updates (turn and flag) could still be problematic