ECE30021/ITP30002 Operating System

Synchronization

(OSC: Ch. 6)

This lecture note is taken from the instructor's resource of Operating System Concept, 9/e and then partly edited/revised by Shin Hong.

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Naïve Concurrent Producer-Consumer

- Bounded Buffer as Circular Queue in Multithreaded Executions

```
Producer(item):

  while (counter == BUFSIZE) ;

  buffer[in] = item;
  in = (in + 1) % BUFSIZE;
  counter++;
```

```
Consumer():

  while (counter == 0) ;

  item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;

  counter--;
  return item ;
```

Race Condition

- `counter++` could be implemented as

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

- `counter--` could be implemented as

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

- Consider this execution interleaving with "count = 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}

- Race condition

multiple processes access (i.e., read and/or write) the same data concurrently, and the outcome depends on the order in which the accesses take place

Critical Section Policy

- Consider a system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has a segment of code as **critical section**
  - A 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 at the same time
- ***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**

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Mechanism for Implementing Critical-Section

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

Critical-Section Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive
  - Preemptive kernel allows preemption of process when running in kernel mode
  - Non-preemptive kernel runs until exits kernel mode, blocks, or voluntarily yields CPU
    - ~~Essentially free of race conditions in kernel mode~~

Peterson's Solution

- Software-based synchronization mechanism
  - Two process solution
  - Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
    - to indicates whose turn it is to enter the critical section
  - **`Boolean flag[2]`**
    - to indicate if a process is ready to enter the critical section
      - **`flag[i] = `*`true`*** implies that process $\mathbf{P_i}$ is ready!

Algorithm

```
bool flag[2] ;
bool turn ;
```

```
thread_0 { //_tid = 0
  …
  flag[_tid] = 1 ;
  turn = !_tid ;
  while (flag[!_tid] &&
         turn == !_tid);

  /*critical section*/

  flag[_tid] = 0;
  …
}
```

```
thread_1 { //_tid = 1
  …
  flag[_tid] = 1 ;
  turn = !_tid ;
  while (flag[!_tid] &&
         turn == !_tid);

  /*critical section*/

  flag[_tid] = 0 ;
  …
}
```

An Example of Wrong Mutual Exclusion

```
char cnt=0,x=0,y=0,z=0;

void process() {
     char me=_pid +1; /* me is 1 or 2*/
again:
        x = me;
        If (y ==0 || y== me) ;
        else goto again;

        z =me;
        If (x == me) ;
        else goto again;

        y=me;
        If(z==me);
        else goto again;

        /* enter critical section */
        cnt++;
        assert( cnt ==1);
        cnt --;
        goto again;
}
```

Software locks

Critical section

Mutual Exclusion Algorithm

Process 0

Process 1

x = 2
If(y==0 || y ==2)
z = 2
If(x==2)

x = 1
If(y==0 || y == 1)

y=2
If (z==2)
cnt++

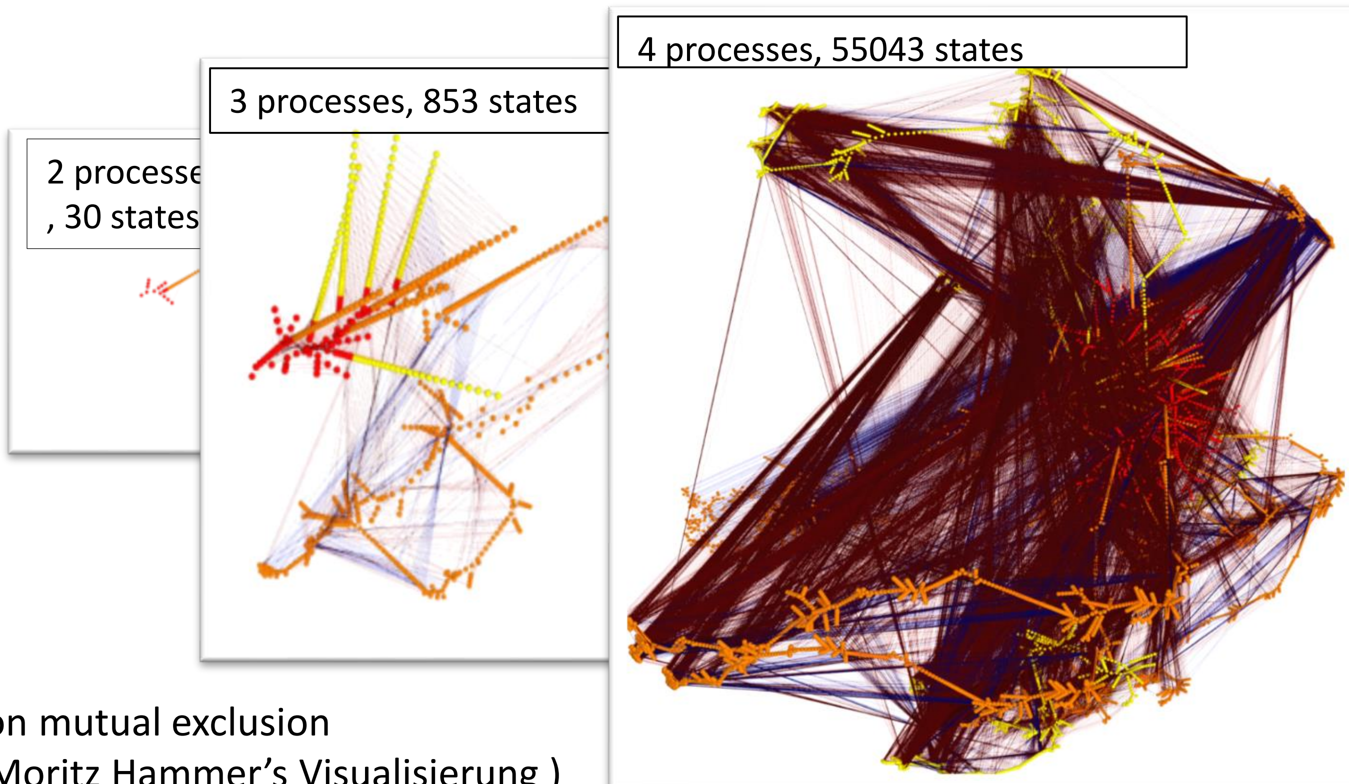
z = 1
If(x == 1)
y = 1
If(z == 1)
cnt++

Violation detected !!!

Counter Example

High Complexity of Peterson's algorithm

- The number of execution paths is exponential to the number of threads and the lengths due to non-deterministic thread scheduling
  - Testing technique for sequential programs do not properly work



Ex. Peterson mutual exclusion
(From Dr. Moritz Hammer's Visualisierung)