Operating Systems

Lecture #8: Critical Sections

Written by David Goodwin
based on the lecture series of Dr. Dayou Li
and the book *Understanding Operating Systems* 4^{th} *ed.*
by *I.M.Flynn and A.McIver McHoes* (2006)

Department of Computer Science and Technology,
University of Bedfordshire.

Operating Systems, 2013

18^{th} March 2013

Outline

CRITICAL SECTIONS

Introduction - a problem

- Problem
  - If a process is updating a data structure and another process is allowed to run before the updating completes, the result may be inconsistent
  - Example 1 (Incorrect result):
    - Two processes A and B both run ++x which contains the following three instructions:
      ld _x, r1
      addi #1, r1
      st r1, _x
    - A is pre-empted by B after the 1st instruction is executed
    - B finished after its three instruction are executed
    - Switch back to A

Introduction - a problem

- Effect: x increments for B
- Net effect: x does not increment for A

	r1	_x
ld _x, r1	(_x)	(_x)
ld _x, r1	(_x)	(_x)
addi #1, r1	(_x+1)	(_x)
st r1, _x	Null	(_x)+1
addi #1, r1	Null+1 = Null	(_x)
st r1, _x	Null	(_x)

CRITICAL SECTIONS

- The common element in all synchronisation schemes is to allow a process to finish work on a critical section
- Also not allowing other processes to have access to it before the critical section of the program has completed.
- This is called a **critical section** because its execution must be handled as a unit.
- The processes within a critical section can't be interleaved without threatening the integrity of the operation.
- Definition
  - Race conditions: the correctness of result depends on the relative timing among two or more processes sharing the same resource
  - Critical section: code embodies race conditions

MANAGEMENT FUNCTIONS

MANAGEMENT FUNCTIONS

- Interrupt control
  - Interrupt: the processor interrupts a process to stop it when the time slot assigned to the process is due to end
  - If interruption occurs at critical section, the problem mentioned earlier takes place
  - Protection of critical sections from being interrupted:
    - Disable interrupts
    - Interrupt priority levels: higher level interrupts are allowed and lower ones are blocked
    - Problems:
    - An erroneous process may never re-enable if it disables interrupt
    - Clock interrupt may be lost
    - Preventing multi-processes waiting for critical sections at the same time

MANAGEMENT FUNCTIONS

- Atomic instructions
  - Atomic instruction is indivisible
  - Instruction ++x is not an atomic instruction as it consists of three instructions
  - Building atomicity of a relative large code section using mutual exclusion (also known as **mutex** lock [MUTual EXclusion])
  - Example 2 (Test-and-set):
    _mutex_lock:
    tas _lock
    blt _mutex_lock
    ret
    - Process A tries to get a lock for a memory location
    - It tests to see if it can get the lock by running tas
    - If the lock is used by another process then it follows branch of less than (blt) to go back loop and try again
    - Otherwise, it gets the lock and returns (ret)

Test-and-set

- Test-and-set (TS)
  - In single machine cycle, tests to see if the "key" is available
  - if it is, sets it to unavailable
  - Key is a single bit in a storage location that is zero (if free) or one (if busy).
    - Process (p1) test the condition code using TS, before entering critical section.
    - if no other process is in the critical section, then p1 may enter, key is set from zero to one (busy)
    - if p1 finds the busy code, then it's placed in a waiting loop, where it continues to test the key.
  - unless first-come first-served was set up, some processes could be favoured over others
  - waiting processes remain in un-productive, resource-consuming wait loops, known as **busy-waiting**

WAIT and SIGNAL

- WAIT and SIGNAL
  - modification of test-and-set, designed to remove busy-waiting
  - WAIT
    - Activated when process encounters a busy condition
    - sets process's process control block to blacked state and links it to the queue of processes waiting
    - process scheduler selects another process for execution
  - SIGNAL
    - Activated when a process exits the critical section and condition code is "free".
    - checks queue of processes waiting to enter and selects one, setting it to READY state
  - WAIT and SIGNAL operations free processes from busy waiting, returning control to the OS.

Peterson's algorithm

- Peterson's algorithm
  - Two competing processes to take turns in using a shared resource, using only shared memory for communication
  - If one is not interested in using the resource, the other can use it even when it is not its turn
  - Example 3 (Peterson's code with processes A and B):

```
int turn;
int want [2];
void mutex_lock (int who)  who test the lock
{
int other;
other = 1 – who;
want [who] = 1;
turn = other;
while (want[other]turn ≠ who);
}
```

Peterson's algorithm

- A and B can never be in the critical section at the same time: If A is in its critical section, then either want[1] is false (meaning B has left its critical section) or turn is 0 (meaning B is just now trying to enter the critical section, but graciously waiting). In both cases, B cannot be in critical section when A is in critical section.
- A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section.
- a process will not wait longer than one turn for entrance to the critical section: After giving priority to the other process, this process will run to completion and set its flag to 0, thereby allowing the other process to enter the critical section.

Semaphores

- Semaphores
  - A semaphore is originally a signalling device used in railway to protect a section of rail track
  - In OS it is a protected variable or abstract data type for restricting access to shared resources such as shared memory in a multi-process environment
  - An OS controls a semaphore by two operations:
    - up(semaphore s):
      s = s+1
    - down(semaphore s):
      s = 0 if s==0,
      s = s-1if s>0

Semaphores

- A **semephore** is a non-negative integer variable used as a flag.
- In OS a semephoe performs a similar function to railway device:
  - it signals if and when a resource is free and can be used by a process
- "up" operation, V(s)
  - fetch, increment, and store sequence
  - like tests-and-set, must be performed as a single indivisible action to avoid deadlocks
  - s cannot be accessed by other processes during the operation
- "down" operation, P(s)
  - test, fetch, decrement, and store sequence
  - like tests-and-set, must be performed as a single indivisible action to avoid deadlocks

Semaphores

- Example 3: Four processes P1 to P4 share the same resource and OS runs a semaphore
  - P3 is placed in WAIT state on state 4
  - states 6 and 8, when a process exits the critical section, value of s is reset to 1 (free), this triggers one of the blocked processes, entering into the critical section, resetting s the 0 (busy).
  - state 7, P1 and P2 are not trying to do processing.

State	calling process	operation	in critical section	blocked	value of s
0					1
1	P1	P(s)	P1		0
2	P1	V(s)			1
3	P2	P(s)	P2		0
4	P3	P(s)	P2	P3	0
5	P4	P(s)	P2	P3, P4	0
6	P2	V(s)	P3	P4	0
7			P3	P4	0
8	P3	V(s)	P4		0
9	P4	V(s)			1

Monitors

- Monitors
  - The idea comes from physical monitors – you can only display something at a time on your computer's monitor
  - Implementing semaphore using monitor
    - When a process occupies the monitor, OS executes down() to block and stop Other processes to access to the monitor
    - When a process is leaves, OS executes up(), to allow one of processes blocked to use the monitor

```
monitor semaphore
{
        unsigned int s;
        void up (void)
        {
                ++ s;
                signal;
        }
        void down (void)
        {
                while (s == 0)
                        wait;
                -- s;
        }
}
```

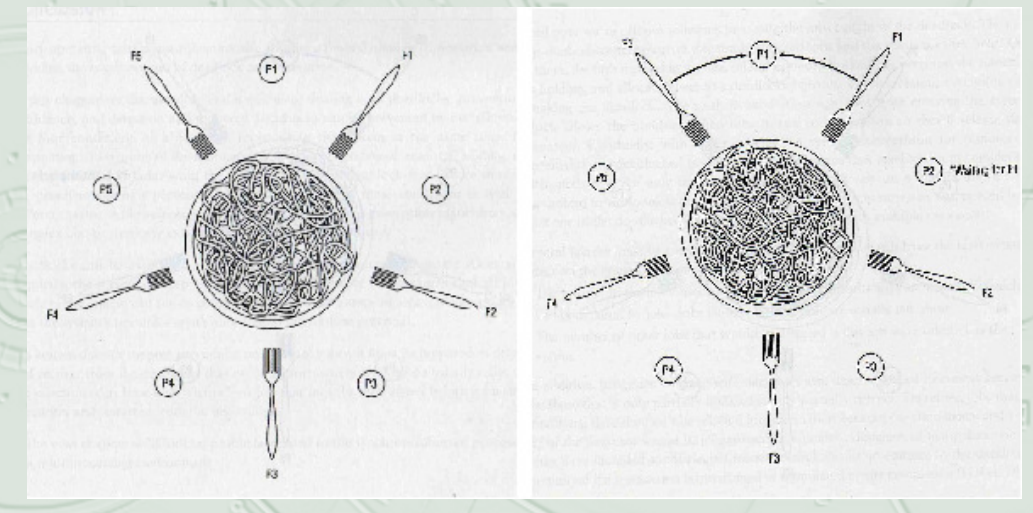
INTERESTING PROBLEMS

INTERESTING PROBLEMS - starvation

- Dining philosophers problem (also known as starvation problem)
- The scenario:
  - Five philosophers sit at a round table and in the centre lies a plate of spaghetti that is available to everyone
  - There are chopsticks on the table – one between each philosopher
  - The philosophers can have three status
    - Think (because they are philosophers)
    - Eat (as they are hungry) and
    - Wait (two chopsticks are need to eat spaghetti but only five available)

INTERESTING PROBLEMS - starvation

- The problem
  - P1 picks C1 and C5 and starts to eat, P3 follows P1 by picking up C2 and C3, P2 has to wait even if he is hungry (because no chopsticks beside him is available
  - P3 finished eating and resuming thinking, P1 is still eating, although C2 becomes available for P2, he is not allowed to pick it up – this is because otherwise it will soon end up with each philosopher holding one chopstick (no one can eat)



INTERESTING PROBLEMS - starvation

- P2 has to wait for C1 which is held by P1
- For some reason, P3 becomes hungry again, so he simply picks up C2 and C3 and eat
- Poor P2 has no chopstick available at all
- Now P1 decides to think and C1 becomes available to P2, however, P2 is not allowed to pick up C1 because C2 is not available
- P2 has to wait for C2 which is held by P3
- As long as P1 and P3 alternate their use of chopsticks (resources) P2 will never have an opportunity to eat

INTERESTING PROBLEMS - starvation

- Controlling philosophers
  - Three groups: thinking (T), hungry (H), and eating (E)
  - If philo i is H and neither of his neighbour is E, then he can become E
  - If he returns to T, OS tests his neighbours to see if they want to change to E
- Code:

```
#define T 1;
#define H 2;
#define E 3;
int philo_state[5];
void begin_eat(int i)
{
    philo_state[i] = H;
    test(i);
}
void end_eat(int i)
{
    philo_state[i] = T;
    test((i+1)%5);
    test((i-1)%5);
}
```

```
void test(int i)
{
    if((philo_state[i] == H) ^
       (philo_state[(i+1)%5]≠ E) ^
       (philo_state[(i-1)%5]≠ E))
    {
          philo_state[i] = E;
    }
}
```

1

INTERESTING PROBLEMS - producer-customer

- Producer-customer problem
  - One process who produces data is called a producer
  - The other process who uses the data is called a customer
  - The producer can produce data at the time when the customer has no intention to use or the amount of data is more than enough for the customer for using at a time
  - Solution
    - Develop a buffer in between the producer and the customer
    - Both can access it but not at the same time

INTERESTING PROBLEMS - producer-customer

- Producer-customer problem
  - CPU can generate output data much faster than a printer can print in
  - need a buffer to temporarily store data from CPU, retrieved at an appropriate speed by the printer
    - buffer can only hold a finite amount of data
    - synchronisation process must delay the CPU from generating more data while the buffer is full (and also preventing the printer from retrieving data when the buffer is empty)
    - Implemented by two counting sememphores, one to indicte number of full positions in buffer, one to indicate the number of empty positions in buffer.
    - a third semephore (mutex) will ensure mutual exclusion

INTERESTING PROBLEMS - producer-customer

- Code

```
int in_queue = 0;
int mutex = 1;
void put(int i)
{
    down(&mutex);
    enqueue(i);
    up(in_queue);
    up(&mutex);
    down(&philo_lock[i]);
}
void get()
{
    int i;;
    down(&in_queue);
    down(&mutes);
    i = dequeue();
    up(&mutex);
}
```