COMP3511 Operating Systems

Topic 6: Process Synchronization (Part I)

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Acknowledgment: The lecture notes are based on various sources on the Internet

Motivation

- Processes can execute concurrently and they may be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanism(s) to ensure the orderly execution of cooperating processes

Illustration of the Problem

- Suppose that we want to provide a solution to the Producer-Consumer problem that fills all the buffers
- We can do so by having an integer `counter` that keeps track of the number of full buffer
- Initially, `counter` is set to 0. It is incremented each time by the producer after it produces an item and places in the buffer and is decremented each time by the consumer after it consumes an item in the buffer

Producer and Consumer Problem

- Producer

```
while(1) {
   // produce an item in next produced
   while(counter == BUFFER_SIZE);
   // do nothing
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

- Consumer

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

Race Condition

It is a undesirable situation where several processes access and manipulate a shared data concurrently and the outcome of the executions depends on the particular order in which the accesses or executions take place

- `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 executes `register1 = counter` (register1 = 5)
  - S1: producer executes `register1 = register1 + 1` (register1 = 6)
  - S2: consumer executes `register2 = counter` (register2 = 5)
  - S3: consumer executes `register2 = register2 - 1` (register2 = 4)
  - S4: producer executes `counter = register1` (counter = 6)
  - S5: consumer executes `counter = register2` (counter = 4)

Critical Section and Critical Section Problem

- Consider a system with n processes $\{p_0, p_1, \ldots, p_{n-1}\}$
- Each process has a Critical Section segment of code
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section

Critical Section

Critical section is a section of code, common to n cooperating processes, in which the processes may be accessing common variables, updating table, writing file, and etc.

Critical Section Problem

Critical section problem is about the protocol(s) design to ensure when one process is in Critical Section, no other may be in its critical section

Critical Section

- General structure of process p_i is

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

- Entry Section: Code requesting entry into the critical section
- Critical Section: Code in which only one process can execute at any one time
- Exit Section: The end of the critical Section, releasing or allowing others in
- Remainder Section: Rest of the code after the critical section

Solution to Critical-Section Problem

1. Mutual Exclusion: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
   i.e., Only one process can be in its critical section at any time
2. Progress: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely
   i.e., No process running outside critical section can block other processes from entering the critical section
3. Bounded Waiting: There exists a bound, or limit, 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
   i.e., No process should wait arbitrarily long to enter its critical section

- Assume that each process executes at a non-zero speed
- There is NO assumption concerning relative speed of n processes

Question

- Which basic operations are sufficient to realize the mentioned requirements for critical sections?
- Answer: Two "atomic" operations are sufficient
  - Reading the value of a variable (load instruction)
  - Writing a value into a variable (store instruction)

Atomic Operation

It is one single non-interrupt able machine instruction. A context switch may happen before or after the instruction is executed but not in between

Example: The following is an atomic instruction

```
turn = j;
```

Solution 1

- In this solution, the processes use a shared global variable `turn` to indicate the next process to enter the critical section. The initial value of turn can be 0 or 1

```
int turn = 0;
```

Process P_0

```
while(1) {
  while(turn != 0);    (1)
  CRITICAL SECTION     (2)
  turn = 1;            (3)
  REMAINDER SECTION    (4)
}
```

Process P_1

```
while(1) {
  while(turn != 1);    (1)
  CRITICAL SECTION     (2)
  turn = 0;            (3)
  REMAINDER SECTION    (4)
}
```

- INCORRECT, because it satisfies Mutual Exclusion, but not Progress
- Explanation
  - P_0: (1), (2), (3), (4) $\rightarrow$ Context switch
  - P_1: (1), (2), (3), (4), (1), (1), (1), ...

Process P_0 does not request access to the critical section, turn will not change back to 1, P_1 is blocked

Solution 2

- In this solution, the processes use a shared global array `intendToEnter` to indicate their intention to enter the critical section

```
int intendToEnter[] = {0, 0}
```

Process P_0

```
while(1) {
  intendToEnter[0] = 1;     (1)
  while(intendToEnter[1]); (2)
  CRITICAL SECTION          (3)
  intendToEnter[0] = 0;     (4)
  REMAINDER SECTION         (5)
}
```

Process P_1

```
while(1) {
  intendToEnter[1] = 1;     (1)
  while(intendToEnter[0]); (2)
  CRITICAL SECTION          (3)
  intendToEnter[1] = 0;     (4)
  REMAINDER SECTION         (5)
}
```

- INCORRECT, because it also satisfies Mutual Exclusion, but not Bounded Waiting
- Explanation:
  - P_0: (1) $\rightarrow$ Context switch, P_1: (1) $\rightarrow$ Context switch
  - P_0: (2) $\rightarrow$ Context switch, P_1: (2)

Each process requests access, but neither process takes access. Both processes block.

Peterson's Algorithm

- Peterson's algorithm is a combination of solutions (1) and (2)
- Assume that the LOAD and STORE instructions are atomic, i.e. cannot be interrupted
- Shared variables

```
int intendToEnter[] = {0, 0}
int turn; // no initial value for turn is needed
```

Process P_0

```
while(1) {
  intendToEnter[0] = 1;          (1)
  turn = 1;                      (2)
  while(intendToEnter[1] &&
        turn == 1);              (3)
  CRITICAL SECTION               (4)
  intendToEnter[0] = 0;          (5)
  REMAINDER SECTION              (6)
}
```

Process P_1

```
while(1) {
  intendToEnter[1] = 1;          (1)
  turn = 0;                      (2)
  while(intendToEnter[0] &&
        turn == 0);              (3)
  CRITICAL SECTION               (4)
  intendToEnter[1] = 0;          (5)
  REMAINDER SECTION              (6)
}
```

Proof

- Mutual exclusion: P_i enters its critical section only if either intendToEnter[j] == 0 or turn == i. If both processes are trying to enter the critical section intendToEnter[0] == intendToEnter[1] == 1, the value of turn can be either 0 or 1 but not both
- P_i can be prevented from entering its critical section only if it is stuck in the while loop with the condition intendToEnter[j] == 1 and turn == j
- If P_j is not ready to enter the critical section, then intendToEnter[j] == 0 and P_i can enter its critical section
- If P_j is inside the critical section, once P_j exits its critical section, it will reset intendToEnter[j] to 0, allowing P_i can enter its critical section. If P_j resets intendToEnter[j] to 1, it must also set turn to i. Thus since P_i does not change the value of the variable turn while executing the while statement, P_i can enter its critical section (progress) after at most one entry (bounded waiting)

Synchronization Hardware

- The solutions to critical section problem presented so far have all been implemented in software
- The critical section problem can be solved if we could disallow interrupts to occur while a shared variable is being modified (atomic instructions)

- However, this is a bit extreme, since it does not just restrict another process that will be modifying the same shared variable from being switch in, it prevents ANY other process from being switched in
- This approach would not work in a multiprocessor environment

- Hardware approaches to solve the critical section problem involve the provision of special hardware instructions, for example
  - Instructions that allow us to test and modify the contents of a word atomically

Hardware Solutions: Test-and-Set

- Test-and-Set instruction: Test and modify the content of a word atomically

```
int test_and_set(int* target) {
    int retVal = *target;
    *target = 1;
    return retVal;
}
```

- Shared variable lock, initialize to 0

```
do {
    while(test_and_set(&lock));          // do nothing
    // critical section
    lock = 0;
    // remainder section
}while(1);
```

Bounded-Waiting Mutual Exclusion with Test-and-Set


```
do {
   waiting[i] = 1;
   key = 1;
   while(waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = 0;
   // critical section
   j = (i + 1) % n;
   while((j != i) && !waiting[j])
      j = (j + 1) % n;
   if(j == i)
      lock = 0;
   else
      waiting[j] = 0;
   // remainder section
}while(1);
```

Sketch Proof

- Mutual exclusion: P_i enters its critical section only if either waiting[i] == 0 or key == 0. The value of key can become 0 only if *test_and_set*() is executed. The first process to execute *test_and_set*() will find key == 0; all others must wait. The variable waiting[i] can become 0 only if another process leaves its critical section; only one waiting[i] is set to 0, maintaining the mutual-exclusion requirement
- Progress: Since a process existing its critical section either sets lock to 0 or sets waiting[j] to 0. Both allow a process that is waiting to enter its critical section to proceed
- Bounded-waiting: When a process leaves its critical section, it scans the array waiting in cyclic order (i+1, i+2, ..., n-1, 0, 1, ..., i-1). It designates the first process in this ordering that is in the entry section (waiting[j] == 1) as the next one to enter the critical section. Any process waiting to enter its critical section will thus to do within n - 1 turns

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- To access the critical regions with it by first `acquire()` a lock then `release()` it
  - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions

```
acquire() {
   while(!available)
      ;                          // busy wait
   available = false;
}
release() {
   available = true;
}
```

```
do {
   acquire lock
      critical section
   release lock
      remainder section
}while(true);
```

Problem

- But this solution requires busy waiting. This lock therefore called a spinlock
  - Spinlock wastes CPU cycles due to busy waiting, but it does have one advantage in that no context switch is required when a process must wait on a lock, and a contest switch may take considerable time. Thus when locks are expected to be held for short times, spinlock is useful
  - Spinlocks are often used in multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor

Semaphores

- Earlier we have seen several low-level synchronization mechanisms
- Now we will study some higher level synchronization mechanisms
- A semaphore is a very general synchronization tool
- A semaphore S is an integer variable together with two standard atomic operations:

wait (also called P()):

```
void wait(S) {
  while(S <= 0);     // busy wait
  S--;
}
```

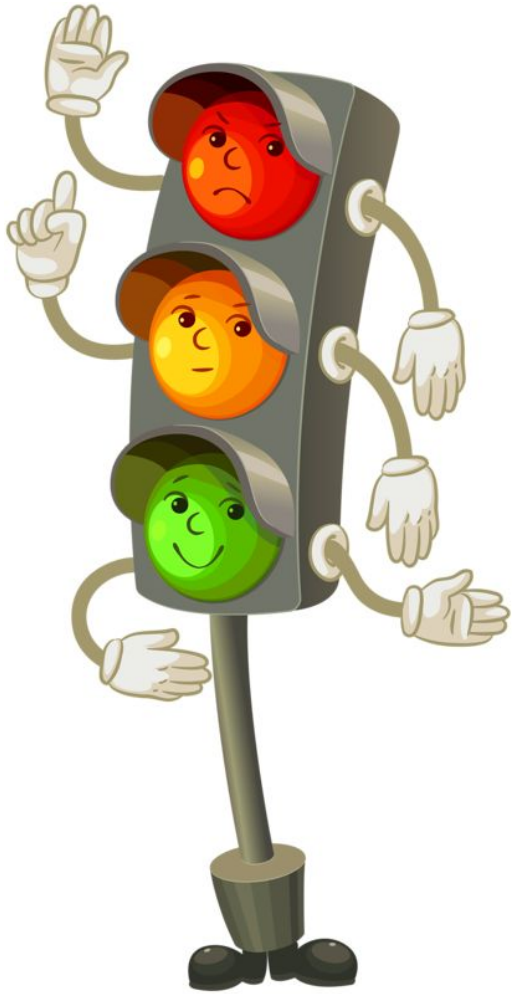
signal (also called V()):

```
void signal(S) {
  S++;
}
```

- The semaphore value modification in the wait and signal operations is executed atomically. We have to guarantee that no more than one process can execute `wait()` and `signal()` operations on the same semaphore at the same time. This is a critical section problem
  - Disable interrupts in a single-processor system would work, but more complicated in a multiprocesssor system

Types of Semaphore

- Counting semaphore
  - Integer value can range over an unrestricted domain
  - It can be used to control access to a given resource consisting of a finite number of instances
  - Semaphore value is initialized to the number of resource available
- Binary semaphore
  - Integer value can range only between 0 and 1
  - This behaves similar to mutex locks



Can implement a counting semaphore S as a binary semaphore

Synchronization with Semaphores

- Suppose two processes P_1 with statement S_1 and P_2 with statement S_2 are executing concurrently
- Suppose S_2 can be executed only after S_1 is executed
- Use a common semaphore `synch` initialized to 0

Process P_1	Process P_2
`...`	`...`
`S1;`	`wait(synch);`
`signal(synch);`	`S2;`
`...`	`...`

- P_2 will execute S_2 only after P_1 has invoked `signal(synch)` after executing S_1

Busy Waiting in Semaphores

- The solutions presented so far all require busy waiting
- For a semaphore when a process calls the `wait` operation, it might have a continuously execute the while loop waiting for the lock
- Busy waiting wastes CPU cycles that could be used more productively
- The semaphore with busy waiting is called spinlock
- Spinlocks are acceptable when the waiting time for the lock is small since we can avoid context switching

"You can see the Doctor now. Don't ask him anything too medical."

Avoid Busy Waiting in Semaphores

To avoid busy waiting, we need to have

- A waiting queue associated with each semaphore and each entry in the waiting queue has two data items
  - value (or type integer)
  - pointer to next record in the list
- Two operations
  - block: place the process invoking the operation on the appropriate waiting queue
  - wakeup: remove one of processes in the waiting queue and place it in the ready queue

- Semaphore values may be negative, whereas this value can never be negative under the classical definition of semaphores with busy waiting
- If a semaphore value is negative, its magnitude is the number of processes waiting on the semaphore

Implementation with no Busy Waiting

- If a process finds that the semaphore value is not positive, it blocks itself
- The blocked process is input in a waiting queue associated with the semaphore
- When the `signal` operation is called by another process, a wakeup operation is initialized and a process from the queue of blocked processes waiting for the semaphore is activated and put into the ready queue

```
struct semaphore {
  int value;
  struct list of process L;
};

wait(semaphore* S) {
  S→value--;
  if(S→value < 0) {
    add this process to S→L;
    block();
  }
}

signal(semaphore* S) {
  S→value++;
  if(S→value <= 0) {
    remove a process P from S→L;
    wakeup(P);
  }
}
```

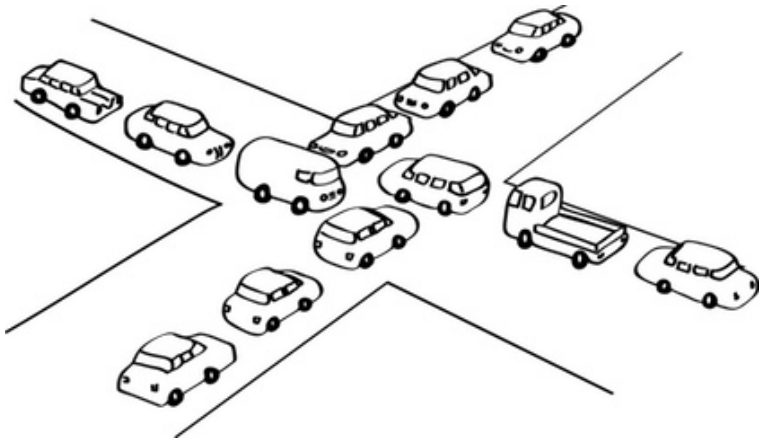
Deadlock and Starvation

Deadlock

It is a situation in which two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
⋮	⋮
signal(S);	signal(Q);
signal(Q);	signal(S);



Starvation (Indefinite Blocking)

It is a situation in which a process may never be removed from the semaphore queue in which it is suspended. For instance, if we remove processes from the queue associated with a semaphore using LIFO (last-in, first-out) order

Priority Inversion

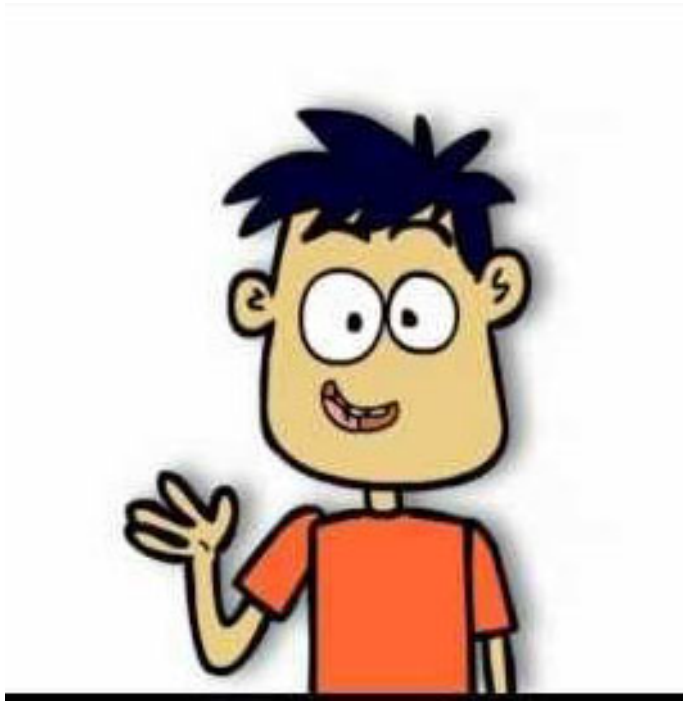
Priority Inversion

A scheduling problem when a lower-priority process holds a lock needed by a higher-priority process

- Situation becomes more complicated if the low-priority process is preempted in favor of another process with a higher priority
- Consider three processes: L, M and H, whose priorities follow the order L<M<H
  - Assume that process H requires resource R, which is currently being accessed by process L
  - Usually process H would wait for process L to finish using resource R
  - Now, suppose M becomes runnable, thereby preempting process L
  - Indirectly, a process with a lower priority (M) has affected how long process H must wait for process L to relinquish resource R
  - This problem is known as priority inversion

Priority Inheritance Protocol

- All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished the resource
- When they are finished, their priorities revert to their original values
  - In the example on the last page, process L would inherit the priority of process H temporarily, thereby preventing process M from preempting its execution
  - Process L relinquish its priority to its original value after finishing using resource R
  - Once resource R is available, process H, not process M would run next



Bounded-Buffer Problem

- n buffers, each can hold one item
- Semaphore `mutex` initialized to 1, semaphore `full` initialized to 0, and semaphore `empty` initialized to n

Producer Process

```
do {
   ...
   // produce an item in
   // next_produced
   ...
   wait(empty);
   wait(mutex);
   ...
   // add next produced to the
   // buffer
   ...
   signal(mutex);
   signal(full);
}while(true);
```

Consumer Process

```
do {
   wait(full);
   wait(mutex);
   ...
   // remove an item from buffer to
   // next_consumed
   ...
   signal(mutex);
   signal(empty);
   ...
   // consume the item in next
   // consumed
   ...
}while(true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers: only read the data set, they do not perform any updates
  - Writers: can both read and write

Problem: Allow multiple readers to read the data set at the same time, while only one single writer can access the shared data at a time

- Several variations of how readers and writers are treated, all involve priorities
- The simplest solution, referred as the first readers-writers problem, requires that no reader be kept waiting unless a writer has already gained access to the shared data
- Shared data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0

Readers-Writers Problem (Cont'd)

Writer Process

```
do {
   wait(rw_mutex);
   ...
   // writing is performed
   ...
   signal(rw_mutex);
}while(true);
```

Reader Process

```
do {
   wait(mutex);
   read_count++;
   if(read_count == 1)
      wait(rw_mutex);
   signal(mutex);
   ...
   // reading is performed
   ...
   wait(mutex);
   read_count--;
   if(read_count == 0)
      signal(rw_mutex);
   signal(mutex);
}while(true);
```

Readers-Writers Problem Variations

- First variation
  - No reader kept waiting unless writer has gained access to use shared object
- Second variation
  - Once writer is ready, it performs write as soon as possible
  - In other word, if a writer is waiting to access the object (implying that there are readers reading at the moment), no new readers may start reading, i.e. they must wait after the writer updates the object
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks, in which multiple processes are permitted to concurrently acquire a reader-writer lock in read mode, but only one process can acquire the reader-writer lock for waiting