CS420: Operating Systems

Process Synchronization

James Moscola
Department of Engineering & Computer Science
York College of Pennsylvania

Based on Operating System Concepts, 9th Edition by Silberschatz, Galvin, Gagne

Background

- **Concurrent access to shared data may result in data inconsistency**
  - Multiple threads/processes changing the same data without synchronization can lead to unpredictable results
- **Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes**
  - Identifying critical sections of code
  - Semaphores (or mutexes) to prevent simultaneous access to shared data

Consider the Following Producer + Consumer

- **Both threads of execution may be trying to access (read or write) the value of counter concurrently**
  - When executed individually, the following code blocks operate correctly
  - When executed concurrently, all bets are off

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

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

Depending on the order of execution, the value of `counter` after these blocks of code run could be several different values

Race Condition

- **At the hardware level, an addition or subtraction may require several steps**

```
counter++

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

```
counter--

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

- **The steps above may get interleaved when executing on a processor and cause incorrect results**
  - Consider the following interleaved execution with `counter=5` initially:

```
T0: producer executes  register1 = counter          { register1 = 5 }
T1: producer executes  register1 = register1 + 1    { register1 = 6 }
T2: consumer executes  register2 = counter          { register2 = 5 }
T3: consumer executes  register2 = register2 - 1    { register2 = 4 }
T4: producer executes  counter   = register1        { counter = 6 }
T5: consumer executes  counter   = register2        { counter = 4 }
```

Race Condition

- **A situation in which several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called a race condition**

Critical Section Problem

- **Consider a system of n processes {P_0, P_1, … P_{n-1}}**
- **Each process has a critical-section segment of code**
  - Process may be changing common variables, updating table, writing file, etc.
    - Example: If two threads modify a shared variable, the portion of code that modifies that variable is a critical-section of code
  - When one process is in its critical-section, no other may be in its critical-section
- **The critical-section problem -- what type of protocol could be implemented that would allow processes to cooperate when accessing shared data/ resources?**
- **Each process must ask permission to enter critical-section of code**

Solution to Critical-Section Problem

- **A solution to the critical section problem must satisfy the following 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 (need to be able to lock sections of code)

  (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

Possible Solutions to the Critical-Section Problem

- **Synchronization Hardware**
  - A hardware-based solution that provides a lock that can be set that only allows a single process to execute in a critical-section

- **Semaphores**
  - Arguably the most common approach for synchronizing access to shared data
  - Programmer friendly

Synchronization Hardware

- **Many systems provide hardware support for critical section code**

- **In a uniprocessors system, could disable interrupts while in critical-section**
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

- **Modern machines provide special atomic hardware instructions that can be used to implement the lock (atomic = non-interruptable)**
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

Critical section of code should be as small as possible, only lock sections that access/modify shared data

Using **TestAndSet** Instruction to Lock

```
do {
  while(TestAndSet(&lock));

  // critical section

  lock = FALSE;

  // remainder section

} while (TRUE);
```

Initialize the value of shared `lock` variable to `FALSE`

Loop continuously until `TestAndSet` returns `FALSE`

```
boolean TestAndSet(boolean *target) {
  boolean rv = *target;
  *target = TRUE;
  return rv;
}
```

Gets current value of `lock` variable from memory, sets value in memory, return previous value from memory

Bounded-Waiting Mutual Exclusion with `TestandSet`

```
do {
  waiting[i] = TRUE;            // Pi is waiting for critical section
  key = TRUE;
  while (waiting[i] && key)     // Pi waiting and doesn't have lock
    key = TestAndSet(&lock);    // Returns FALSE when Pi gets lock
  waiting[i] = FALSE;           // Pi is no longer waiting :-)

  // critical section

  j = (i + 1) % n;                  // Scan waiting array to find
  while ((j != i) && !waiting[j])   // others waiting for lock
    j = (j + 1) % n;

  if (j == i)               // if looped all the way back to self,
    lock = FALSE;           // no one else waiting, release lock
  else                      // otherwise, keep lock set to TRUE
    waiting[j] = FALSE;     // and just tell next P to stop waiting

  // remainder section
  } while (TRUE);
```

Semaphores

- **A synchronization tool that does not require busy waiting (a while loop that runs until conditions are satisfied)**

- **A semaphore is represented as an integer variable**

- **Two standard atomic operations are used to modify a semaphore**
  - **`wait()`** - attempts to decrement the value of the semaphore
  - **`signal()`** - increments the value of the semaphore

Semaphores

- **Two different kinds of semaphores**

  (1) Binary Semaphore

  - Integer value can be either 0 or 1
  - Also called a mutex lock
  - Provides mutual exclusion

  (2) Counting Semaphore

  - Integer value can range over an unrestricted domain
  - Can be used to control access to a system resource with a finite number of instances
    - Initialize to the number of resources available
    - Decrement each time `wait()` is called
      - When value of semaphore reaches 0, no more resources are available
    - Increment each time `signal()` is called

Semaphore as Synchronization Tool

- **Provides mutual exclusion to critical section of code**

```
Semaphore mutex;     // initialized to 1

do {
  wait(mutex);       // acquire lock

  // Critical section

  signal(mutex);     // release lock

  // remainder section

} while(TRUE);
```

Semaphore Implementation

- **Must guarantee that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time**
  - Semaphore value is a shared value that only one process should be able to access and modify at any given time

- **Thus, implementation becomes a critical section problem where the code for `wait()` and `signal()` become a critical section**
  - Could implement with busy waiting since `wait()` and `signal()` code is very short (simply increments or decrements semaphore value)
  - Busy waiting wastes CPU cycles

Semaphore Implementation with No Busy Waiting

- **Associate a wait queue with each semaphore**
- **Each semaphore has two data items:**
  - A value (of type integer) that can be incremented/decremented by `wait/signal`
  - A pointer to a list of PCBs waiting for the semaphore

```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```

- **Two new system calls facilitate semaphore functionality without busy waiting:**
  - **`block()`** – suspends the process that invokes it
  - **`wakeup(P)`** – resumes the execution of a blocked process P (i.e. returns it to the ready queue)

Semaphore Implementation with No Busy Waiting

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

A process calls `wait()` when it wants access to a critical region or resource. If resource is unavailable, then the process blocks.

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

Some other process will eventually call `signal()` to release the semaphore (hopefully). If another process was waiting, it will be removed from the waiting list and added to the ready queue.

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Example: let `S` and `Q` be two semaphores initialized to 1

```
  P0                P1
---------------------
wait(S)       wait(Q)
wait(Q)       wait(S)
  .                .
  .                .
  .                .
signal(S)     signal(Q)
signal(Q)     signal(S)
```

What happens when P_0 executes `wait(S)` and P_1 executes `wait(Q)`?

Problems with Semaphores

- **Incorrect use of semaphore operations (i.e. bad programming):**
  - `signal(mutex)` … `wait(mutex)` -- wrong order
  - `wait(mutex)` … `wait(mutex)` -- decrement TWO resources
  - Omitting of `wait(mutex)` or `signal(mutex)` (or both) -- UGH

Deadlock and Starvation

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended

- **Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process**
  - Solved via priority-inheritance protocol whereby a lower priority process inherits the priority of another higher priority process that needs access to the resources held by the lower priority process