Unit OS3: Concurrency

3.1. Concurrency, Critical Sections, Semaphores

Windows Operating System Internals - by David A. Solomon and Mark E. Russinovich with Andreas Polze

Copyright Notice

Roadmap for Section 3.1.

- The Critical-Section Problem
- Software Solutions
- Synchronization Hardware
- Semaphores
- Synchronization in Windows & Linux

The Critical-Section Problem

- *n* threads all competing to use a shared resource (i.e.; shared data)
- Each thread has a code segment, called *critical section*, in which the shared data is accessed
- Problem:
  Ensure that when one thread is executing in its critical section, no other thread is allowed to execute in its critical section

Solution to Critical-Section Problem

1. Mutual Exclusion
   - Only one thread at a time is allowed into its critical section, among all threads that have critical sections for the same resource or shared data.
   - A thread halted in its non-critical section must not interfere with other threads.
2. Progress
   - A thread remains inside its critical section for a finite time only.
   - No assumptions concerning relative speed of the threads.
3. Bounded Waiting
   - It must no be possible for a thread requiring access to a critical section to be delayed indefinitely.
   - When no thread is in a critical section, any thread that requests entry must be permitted to enter without delay.

Initial Attempts to Solve Problem

- Only 2 threads, T_0 and T_1
- General structure of thread T_i (other thread T_j)

```
do {
   enter section
     critical section
   exit section
     reminder section
} while (1);
```

- Threads may share some common variables to synchronize their actions.

First Attempt: Algorithm 1

- Shared variables - initialization

```
int turn = 0;
```

- `turn == i` $\Rightarrow T_i$ can enter its critical section
- Thread T_i

```
do {
  while (turn != i) ;
    critical section
  turn = j;
    reminder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

Second Attempt: Algorithm 2

- Shared variables - initialization

```
int flag[2]; flag[0] = flag[1] = 0;
```

- `flag[i] == 1` $\Rightarrow T_i$ can enter its critical section
- Thread T_i

```
do {
      flag[i] = 1;
      while (flag[j] == 1) ;
                  critical section
      flag[i] = 0;
            remainder section
} while(1);
```

- Satisfies mutual exclusion, but not progress requirement.

Third Attempt: Algorithm 3
(Peterson's Algorithm - 1981)

- Shared variables of algorithms 1 and 2 - initialization:

```
int flag[2]; flag[0] = flag[1] = 0;
int turn = 0;
```

- Thread T_i

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

- Solves the critical-section problem for two threads.

Dekker's Algorithm (1965)

- This is the first correct solution proposed for the two-thread (two-process) case.
- Originally developed by Dekker in a different context, it was applied to the critical section problem by Dijkstra.
  - Dekker adds the idea of a favored thread and allows access to either thread when the request is uncontested.
  - When there is a conflict, one thread is favored, and the priority reverses after successful execution of the critical section.

Dekker's Algorithm (contd.)

- Shared variables - initialization:

```
int flag[2]; flag[0] = flag[1] = 0;
int turn = 0;
```

- Thread T_i

```
do {
        flag[i] = 1;

        while (flag[j] )
                if (turn == j) {
                        flag[i] = 0;
                        while (turn == j);
                        flag[i] = 1;
                }

                        critical section

        turn = j;
        flag[I] = 0;;

                        remainder section

} while (1);
```


Bakery Algorithm

(Lamport 1979)

A Solution to the Critical Section problem for n threads

- Before entering its critical section, a thread receives a number. Holder of the smallest number enters the critical section.
- If threads T_i and T_j receive the same number, if i < j, then T_i is served first; else T_j is served first.
- The numbering scheme generates numbers in monotonically non-decreasing order; i.e., 1,1,1,2,3,3,3,4,4,5...

Bakery Algorithm

- Notation "<" establishes lexicographical order among 2-tuples (ticket #, thread id #)

  $(a,b) < (c,d)$ if $a < c$ or if $a == c$ and $b < d$

  max $(a_0,\ldots, a_{n-1})$ = { k | $k \geq a_i$ for $i = 0,\ldots, n - 1$ }

- Shared data

```
int choosing[n];
int number[n];
```

  - the ticket

  Data structures are initialized to 0

Bakery Algorithm

```
do {
   choosing[i] = 1;
   number[i] = max(number[0],number[1] ...,number[n-1]) + 1;
   choosing[i] = 0;
   for (j = 0; j < n; j++) {
        while (choosing[j] == 1) ;
        while ((number[j] != 0) &&
                ((number[j],j) ''<'' (number[i],i)));
   }
        critical section
   number[i] = 0;
        remainder section
} while (1);
```

Mutual Exclusion - Hardware Support

- Interrupt Disabling
  - Concurrent threads cannot overlap on a uniprocessor
  - Thread will run until performing a system call or interrupt happens
- Special Atomic Machine Instructions
  - Test and Set Instruction - read & write a memory location
  - Exchange Instruction - swap register and memory location
- Problems with Machine-Instruction Approach
  - Busy waiting
  - Starvation is possible
  - Deadlock is possible

Synchronization Hardware

- Test and modify the content of a word atomically

```
boolean TestAndSet(boolean &target) {
  boolean rv = target;
  target = true;

  return rv;
}
```

Mutual Exclusion with Test-and-Set

- Shared data:

```
boolean lock = false;
```

- Thread T_i

```
do {
  while (TestAndSet(lock)) ;
    critical section
  lock = false;
    remainder section
}
```


Synchronization Hardware

- Atomically swap two variables.

```
void Swap(boolean &a, boolean &b) {
  boolean temp = a;
  a = b;
  b = temp;
}
```

Mutual Exclusion with Swap

- Shared data (initialized to **0**):

```
int lock = 0;
```

- Thread T_i

```
int key;
do {
    key = 1;
    while (key == 1) Swap(lock,key);
         critical section
    lock = 0;
         remainder section
}
```


Semaphores

- Semaphore *S* – integer variable
- can only be accessed via two atomic operations

```
wait (S):
      while (S <= 0);
      S--;

signal (S):
      S++;
```

Critical Section of n Threads

- Shared data:

```
semaphore mutex; //initially mutex = 1
```

- Thread T_i:

```
do {
      wait(mutex);
            critical section
      signal(mutex);
            remainder section
} while (1);
```


Semaphore Implementation

- Semaphores may suspend/resume threads
  - Avoid busy waiting
- Define a semaphore as a record

```
typedef struct {
   int value;
   struct thread *L;
} semaphore;
```

- Assume two simple operations:
  - **suspend()** suspends the thread that invokes it.
  - **resume(*T*)** resumes the execution of a blocked thread **T**.

Implementation

- Semaphore operations now defined as

wait(*S*):
```
    S.value--;
    if (S.value < 0) {
            add this thread to S.L;
            suspend();
    }
```
signal(*S*):
```
    S.value++;
    if (S.value <= 0) {
            remove a thread T from S.L;
            resume(T);
    }
```


Semaphore as a General Synchronization Tool

- Execute *B* in T_j only after *A* executed in T_i
- Use semaphore *flag* initialized to 0
- Code:

T_i	T_j
…	…
A	*wait*(*flag*)
signal(*flag*)	*B*

Two Types of Semaphores

- *Counting* semaphore
  - integer value can range over an unrestricted domain.
- *Binary* semaphore
  - integer value can range only between 0 and 1;
  - can be simpler to implement.
- Counting semaphore *S* can be implemented as a binary semaphore.

Deadlock and Starvation

- **Deadlock** – two or more threads are waiting indefinitely for an event that can be caused by only one of the waiting threads.
- Let *S* and *Q* be two semaphores initialized to 1

T_0	T_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
…	…
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- **Starvation** – indefinite blocking. A thread may never be removed from the semaphore queue in which it is suspended.
- Solution - all code should acquire/release semaphores in same order

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems.
- Uses *spinlocks* on multiprocessor systems.
- Provides *dispatcher objects* which may act as mutexes and semaphores.
- Dispatcher objects may also provide *events*. An event acts much like a condition variable.

Linux Synchronization

- Kernel *disables interrupts* for synchronizing access to global data on uniprocessor systems.
- Uses *spinlocks* for multiprocessor synchronization.
- Uses *semaphores* and *readers-writers* locks when longer sections of code need access to data.
- Implements POSIX synchronization primitives to support multitasking, multithreading (including real-time threads), and multiprocessing.

Further Reading

- Ben-Ari, M., Principles of Concurrent Programming, Prentice Hall, 1982
- Lamport, L., The Mutual Exclusion Problem, Journal of the ACM, April 1986
- Abraham Silberschatz, Peter B. Galvin, Operating System Concepts, John Wiley & Sons, 6th Ed., 2003;
  - Chapter 7 - Process Synchronization
  - Chapter 8 - Deadlocks