Lecture 7: Synchronization

Prof. Alan Mislove (amislove@ccs.neu.edu)
Background

Concurrent access to shared data may result in data inconsistency

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

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 count that keeps track of the number of full buffers. Initially, count 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.
Producer

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

Consumer

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

`counter++` could be implemented as

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

`counter--` could be implemented as

```java
register2 = counter
register2 = register2 - 1
count = 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` {count = 6}
- **S5**: consumer execute `counter = register2` {count = 4}
Generalization: Critical Section Problem

Consider system of n processes \{p_0, p_1, \ldots p_{n-1}\}

Each process has *critical section* segment of code

- 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

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*

Especially challenging with preemptive kernels
Critical Section

General structure of process $p_i$ is

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

Figure 6.1 General structure of a typical process $R$. 
Reqs. for 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.

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.
Peterson’s Solution

Two process solution

Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted

The two processes share two variables:

```c
int turn;
Boolean flag[2];
```

The variable `turn` indicates whose turn it is to enter the critical section

The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process `P_i` is ready!
Algorithm for Process $P_i$

do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j) {}  
    critical section
    flag[i] = FALSE;
    remainder section
} while (TRUE);

Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met
Synchronization Hardware

Many systems provide hardware support for critical section code

Uniprocessors – could disable interrupts
   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
   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);
TestAndSet Instruction

Definition:

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

Shared boolean variable lock, initialized to FALSE

Solution:

```c
    do {
        while ( TestAndSet (&lock ))
            ; // do nothing

        // critical section
        lock = FALSE;

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

Definition:

```c
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```
Solution using Swap

Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

Solution:

```c
do {
    key = TRUE;
    while (key == TRUE) {
        Swap (&lock, &key );
    }
    //    critical section
    lock = FALSE;
    //      remainder section
} while (TRUE);
```
Bounded-waiting Mutual Exclusion with TestAndSet()

do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // critical section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

    // remainder section
} while (TRUE);
Semaphore

Synchronization tool that does not (necessarily) require busy waiting

Semaphore \( S \) – integer variable

Two standard operations modify \( S \): \texttt{wait()} and \texttt{signal()}

Originally called \texttt{P()} and \texttt{V()}

Less complicated

Can only be accessed via two indivisible (atomic) operations

\begin{verbatim}
wait (S) {
    while S <= 0
        ; // no-op
    S--;
}

signal (S) {
    S++;
}
\end{verbatim}
Semaphore as General Synchronization Tool

**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

Also known as **mutex locks**

Can implement a counting semaphore $S$ as a binary semaphore

Provides mutual exclusion

```c
Semaphore mutex;    // initialized to 1

do {
    wait (mutex);
    // Critical Section
    signal (mutex);
    // 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.

Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.

Could now have **busy waiting** in critical section implementation.

- But implementation code is short.
- Little busy waiting if critical section rarely occupied.

Note that applications may spend lots of time in critical sections and therefore this is not a good solution.
Semaphore Implementation without busy waiting

With each semaphore there is an associated waiting queue

Each entry in a waiting queue has two data items:
- value (of 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 Implementation without busy waiting

Implementation of wait:

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

Implementation of signal:

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

\( N \) buffers, each can hold one item

Semaphore \texttt{mutex} initialized to the value 1

Semaphore \texttt{full} initialized to the value 0

Semaphore \texttt{empty} initialized to the value \( N \)
Bounded Buffer Problem (Cont.)

The structure of the producer process

```c
do  {
    // produce an item
    wait (empty);
    wait (mutex);
    // add the item to the buffer
    signal (mutex);
    signal (full);
} while (TRUE);
```
Bounded Buffer Problem (Cont.)

The structure of the consumer process

```c
do {
    wait (full);
    wait (mutex);
    // remove an item from buffer
    signal (mutex);
    signal (empty);
    // consume the item
}
```

while (TRUE);

Monitors

A high-level abstraction that provides a convenient and effective mechanism for process synchronization

*Abstract data type*, internal variables only accessible by code within the procedure

Only one process may be active within the monitor at a time

But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

    procedure Pn (...) {......}

    Initialization code (...) { ... }

}
```
Problems with synchronization

**Deadlock** – 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

- $P_0$
  - wait ($S$);
  - wait ($Q$);
  - .
  - .
  - .
  - signal ($S$);
  - signal ($Q$);

- $P_1$
  - wait ($Q$);
  - wait ($S$);
  - .
  - .
  - .
  - signal ($Q$);
  - signal ($S$);

**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**