Chapter 8

Basic Synchronization Principles
Need for Synchronization

- Multiprogramming
  - Multiple concurrent, independent processes
  - Those processes might want to coordinate activities

\[
\text{shared } x, y
\]

Proc A {
  while (true) {
    <compute A1>
    write(x)
    <compute A2>
    read(y)
  }
}

Proc B {
  while (true) {
    \text{read}(x)
    <compute B1>
    write(y)
    <compute B2>
  }
}

- Clearly, synchronization is needed if
  - A wants B to read \text{x} \text{ after} it writes it & \text{before} it re-writes
Barriers to providing synchronization

- What are the barriers to providing good synchronization capabilities?
  - No widely accepted parallel programming languages
    - CSP
    - Linda
  - No widely use paradigm
    - How do you decompose a problem?
  - OS only provides minimal support
    - Test and Set
    - Semaphore
    - Monitor
shared float balance;

/* Code schema for p1 */
.. balance = balance + amount;
.. /* Schema for p1 */
   load R1, balance
   load R2, amount
   add R1, R2
   store R1, balance
/* Code schema for p2 */
.. balance = balance - amount;
.. /* Schema for p2 */
   load R1, balance
   load R2, amount
   sub R1, R2
   store R1, balance
Critical Section Problem...

/* Schema for p1 */

1. load R1, balance
2. load R2, amount
3. add R1, R2
4. store R1, balance

/* Schema for p2 */

1. load R1, balance
2. load R2, amount
3. sub R1, R2
4. store R1, balance

- Suppose:
  - Execution sequence: 1, 2, 3
    - Lost update: 2
  - Execution sequence: 1, 4, 3, 6
    - Lost update: 3
- Together => non-determinacy
- Race condition exists
Race Condition Example 2

Two processes want to access shared memory at same time

Taken from Modern Operating Systems, 2\textsuperscript{nd} Ed, Tanenbaum, 2001
Using Shared Global Variables – Ver 1

```
Shared integer processnumber = 1;

void processone;
{
    while (true)  
    {
        while (processnumber == 2)
        {
            criticalsectionone;
            processnumber = 2;
            otherstuffone;
        }
    }
}

void processtwo;
{
    while (true)
    {
        while (processnumber == 1)
        {
            criticalsectiontwo;
            processnumber = 1;
            otherstufftwo;
        }
    }
}
```

Single global variable forces **lockstep synchronization**
Using Shared Global Variables – Ver 2

Shared boolean p1inside = false, p2inside = false;

void processone;
{
    while (true) {
        while (p2inside);
        p1inside = true;
        criticalsectionone;
        p1inside = false;
        otherstuffone;
    }
}

void processtwo;
{
    while (true) {
        while (p1inside);
        p2inside = true;
        criticalsectiontwo;
        p2inside = false;
        otherstufftwo;
    }
}

- Process 1 & 2 can both be in the critical sections at the same time
  Because Test & Set operations are not atomic

  ==> Move setting of p1inside/p2inside before test
void processone;
{
    while (true) {
        p1wantsin = true;
        while (p2wantsin)
        {
            
        }
        criticalsectionone;
        p1wantsin = false;
        otherstuffone;
    }
}

void processtwo;
{
    while (true) {
        p2wantsin = true;
        while (p1wantsin)
        {
        }
        criticalsectiontwo;
        p2wantsin = false;
        otherstufftwo;
    }
}

- **Deadlock** can occur if both sets flag at the same time

  ==> Need a way to break out of loops.....
Using Shared Global Variables – Peterson

Shared boolean p1wantsin = false, p2wantsin = false;
Shared int will_wait;

void processone;
{
    while (true) {
        p1wantsin = true;
        will_wait = 1;
        while (p2wantsin && (will_wait == 1))
            ;
        criticalsectionone;
        p1wantsin = false;
        otherstuffone;
    }
}

void processtwo;
{
    while (true) {
        p2wantsin = true;
        will_wait = 2;
        while (p1wantsin && (will_wait == 2))
            ;
        criticalsectiontwo;
        p2wantsin = false;
        otherstufftwo;
    }
}

• Guarantees mutual exclusion and no blocking
Wherein Lies the Problem?

- Problem stems from interruption of software-based process while executing critical code (low-level)
- Solution
  - Identify critical section
  - *Disable interrupts* while in Critical Section

```c
/* Program for P1 */
DisableInterrupts();
balance = balance + amount; //CS
EnableInterrupts();

/* Program for P2 */
DisableInterrupts();
Balance = balance - amount; //CS
EnableInterrupts();
```

shared double balance;
Using Interrupts...

- This works *BUT*...
  - Allows process to disable interrupts for arbitrarily long time
  - What if I/O interrupt needed?
  - What if one of the processes is in infinite loop inside the Critical Section

- Let’s examine the use of Shared Variables again....
/* Program for P1 */
.. 
/* Acquire lock */
while(lock) {NULL;}
lock = TRUE;
/* Execute critical section */
balance = balance + amount;
/* Release lock */
lock = FALSE;
..

/* Program for P2 */
.. 
/* Acquire lock */
while(lock) {NULL;}
lock = TRUE;
/* Execute critical section */
balance = balance - amount;
/* Release lock */
lock = FALSE;
..

lock == FALSE
  => No process in CS
  => Any process can enter CS

lock == TRUE
  => One process in CS
  => No other process admitted to CS
Synchronizing Variable...

- What if P1 interrupted after lock Set to TRUE
  => P2 cannot execute past while does hard wait
  => Wasted CPU time

- What if P1 interrupted after Test, before Set
  => P1 & P2 can be in the CS at the same time !!!

- Wasted CPU time is bad, but tolerable.....
  Critical Section Violation cannot be tolerated
  ==> Need Un-interruptable “Test & Set” operation
Un-interruptible Test & Set

\textbf{enter}(\text{lock}) \{ \\
\text{disableInterrupts}(); \\
/* Loop until lock TRUE */ \\
\text{while} \ (\text{lock}) \{ \\
\text{/* Let interrupts occur */} \\
\text{enableInterrupts}(); \uparrow \\
\text{disableInterrupts}(); \\
\} \\
\text{lock} = \text{TRUE}; \\
\text{enableInterrupts}(); \\
\}

\textbf{exit}(\text{lock}) \{ \\
\text{disableInterrupts}(); \\
\text{lock} = \text{FALSE}; \\
\text{enableInterrupts}(); \\
\}

Enable interrupts so that the OS, I/O can use them

Re-disable interrupts when ready to test again
Un-interruptible Test & Set...

- **Solution**

  P1
  
  ```
  enter(lock);
  CS { balance = balance + amount; }
  exit(lock);
  ```

  P2
  
  ```
  enter(lock);
  CS { balance = balance - amount; }
  exit(lock);
  ```

- **Note**
  - CS is totally bounded by enter/exit
  - P2 can still wait (wasted CPU cycles) if P1 is interrupted after setting lock (i.e., entering critical section), but
  - **Mutual exclusion is achieved!!!!!**

- Does not generalize to multi-processing
Protecting Multiple Components

Shared: list L,
boolean ListLK = False;
boolean LngthLK = False;

/* Program for P1 */
enter(listLK);
<delete element>;
exit(listLK);

<intermediate comp.>;
enter(lngthLK);
<update length>;
exit(lngthLK);

/* Program for P2 */
enter(lngthLK);
<update length>;
exit(lngthLK);

<intermediate comp.>;
enter(listLK);
<delete element>;
exit(listLK);

- Use enter/exit to update structure with 2 pieces if information
- \textit{But try to minimize time component locked out}
Protecting Multiple Components: 1st try

Shared: list L,
    boolean ListLK = False;
    boolean LngthLK = False;

/* Program for P1 */
enter(listLK);
    <delete element>;
exit(listLK);

<intermediate comp.>;
enter(lngthLK);
    <update length>;
exit(lngthLK);

/* Program for P2 */
enter(lngthLK);
    <update length>;
exit(lngthLK);

<intermediate comp.>;
enter(listLK);
    <delete element>;
exit(listLK);

Suppose: P1... ; P2 runs & finishes; P1 ......
Any access to Lnth vble during “intermediate comp.” will be incorrect !!!
=> Programming Error: List and variable need to be updated together
Protecting Multiple Components: 2nd try

Shared: list L,
      boolean ListLK = False;
      boolean LngthLK = False;

/* Program for P1 */

enter(listLK);
<delete element>;
<intermediate comp.>;
enter(lngthLK);
<update length>;
exit(listLK);
exit(lngthLK);

/* Program for P2 */

enter(lngthLK);
<update length>;
<intermediate comp.>;
enter(listLK)
<delete element>;
exit(lngthLK);
exit(listLK);

Suppose: P1...∩ ⊗ ;
P2 runs to ⊗ and blocks ;
P1 starts & blocks on “enter”
=> DEADLOCK

CS 3204 - Arthur
Deadlock

- When 2 or more processes get into a state whereby each is holding a resource requested by the other

P1
  .
Request Resource$_1$
  .
Request Resource$_2$

P2
  .
Request Resource$_2$
  .
Request Resource$_1$

P1 requests and gets R$_1$
interrupt
P2 requests and gets R$_2$
interrupt
P1 requests R$_2$ and blocks
P2 requests R$_1$ and blocks
Solution to Synchronization

- The previous examples have illustrated 2 methods for synchronizing / coordinating processes
  - Interrupt
  - Shared variable

- Each has its own set of problems
  - Interrupt
    - May be disabled for too long
  - Shared variable
    - Test, then set – interruptible
    - Non-interruptible – gets complex

- Dijkstra introduces a 3\textsuperscript{rd} and much more preferable method
  - Semaphore
Semaphore

- Dijkstra, 1965

- Synchronization primitive with no busy waiting

- It is an integer variable changed or tested by one of the two indivisible operations

- Actually implemented as a protected variable type

  \[ \text{var } x : \text{ semaphore} \]
Semaphore operations

- **P** operation ("wait")
  - Requests permission to use a critical resource
    
    
    $$S = S - 1;$$
    
    if 
    
    $$S < 0$$ then
    
    put calling process on queue

- **V** operation ("signal")
  - Releases the critical resource
    
    $$S = S + 1;$$
    
    if 
    
    $$S <= 0$$ then
    
    remove one process from queue

- Queues are associated with each semaphore variable
Semaphore : Example

Critical resource \( T \)
Semaphore \( S \leftarrow \text{initial\_value} \)
Processes \( A, B \)

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \cdot )</td>
<td>( \cdot )</td>
</tr>
<tr>
<td>( P(S); )</td>
<td>( P(S); )</td>
</tr>
<tr>
<td>(&lt;\text{CS}&gt; \ /* \text{access} \ T \ */ )</td>
<td>(&lt;\text{CS}&gt; \ /* \text{access} \ T \ */ )</td>
</tr>
<tr>
<td>( V(S); )</td>
<td>( V(S); )</td>
</tr>
<tr>
<td>( \cdot )</td>
<td>( \cdot )</td>
</tr>
</tbody>
</table>
Semaphore: Example...

\[ \text{var } S : \text{ semaphore } \leftarrow 1 \]

Queue associated with \( S \)

|   |   |   |   |   |   |   |

Value of \( S \): 1

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{P}(S); \newline &lt;\text{cs}&gt; \newline \text{V}(S);</td>
<td>\text{P}(S); \newline &lt;\text{cs}&gt; \newline \text{V}(S);</td>
<td>\text{P}(S); \newline &lt;\text{cs}&gt; \newline \text{V}(S);</td>
</tr>
</tbody>
</table>
Types of Semaphores

- Binary Semaphores
  - Maximum value is 1

- Counting Semaphores
  - Maximum value is greater than 1

- Both use same P and V definitions

- Synchronizing code and initialization determines what values are needed, and therefore, what kind of semaphore will be used
Using Semaphores

Shared semaphore \texttt{mutex} = 1;

\begin{verbatim}
proc_1() {
    while(true) {
        <compute section>
        P(mutex);
        <critical section>
        V(mutex);
    }
}

proc_2() {
    while(true) {
        <compute section>
        P(mutex);
        <critical section>
        V(mutex);
    }
}
\end{verbatim}

(1) P1 \implies P(mutex)
Decrements; < 0 ?; NO (0);
P1 Enters CS;
P1 interrupted

(2) P2 \implies P(mutex)
Decrements; < 0 ?; YES (-1)
P2 \textbf{blocks} on \texttt{mutex}

(3) P1 finishes CS work
P1 \implies V(mutex);
Increments; \leq 0 ?; YES (0)
P2 woken & proceeds

Non-Interruptable "Test & Sets"
Using Semaphores - Example 1

Shared semaphore mutex = 1;

proc_0() {
    ...
P(mutex);
    balance = balance + amount;
V(mutex);
    ...
}

proc_1() {
    ...
P(mutex);
    balance = balance - amount;
V(mutex);
    ...
}

Suppose P1 issues P(mutex) first ......  
No Problem

Suppose P2 issues P(mutex) first ......

Note: Could use Interrupts to implement solution,
But (1) with interrupts masked off, what happens if
a prior I/O request is satisfied
(2) Interrupt approach would not work on Multiprocessor
Using Semaphores – Example 2

Shared semaphore: s1 = 0, s2 = 0;  

Note: values started at 0... ok?

```
proc_A() {
    while(true) {
        <compute A1>
        write(x);
        V(s1);
        <compute A2>;
        P(s2);
        read(y);
    }
}
```

```
proc_B() {
    while(true) {
        P(s1);
        read(x);
        <compute B1>;
        write(y);
        V(s2);
        <compute B2>;
    }
}
```

- Cannot use Interrupt disable/enable here because we have *multiple distinct synchronization points*
- Interrupt disable/enable can only distinguish 1 synchronization event
- **Therefore, 2 Semaphores**
Using Hardware Test & Set [TS(s)] to Implement Binary Semaphore “Semantics”

boolean s = FALSE;
...
while( TS(s) );
<critical section>
S = FALSE;
...

■ TS(s)
  ■ Test s
  ■ Set s to True
  ■ Return original value

semaphore s = 1;
...
?
≡
P(s);
<critical section>
V(s);
...

Note: No actual queueing, each process just “hard waits”
Counting Semaphores

- Most of our examples have only required Binary Semaphore
  - Only 0 or 1 values

- But synchronization problems arise that require a more general form of semaphores

- Use counting semaphores
  - Values: non-negative integers
Classical Problems

- Producer / Consumer Problem

- Readers – Writers Problem
Producer / Consumer Problem (Classic)

- Critical resource
  - Set of message buffers

- 2 Processes
  - Producer: Creates a message and places it in the buffer
  - Consumer: Reads a message and deletes it from the buffer

- Objective
  - Allow the producer and consumer to run concurrently
P/C...

- **Constraints**
  - Producer must have a non-full buffer to put its message into
  - Consumer must have a non-empty buffer to read
  - Mutually exclusive access to Buffer pool

- **Unbounded Buffer problem**
  - Infinite buffers
  - Producer never has to wait
  - Not interesting nor practical

- **Bounded Buffer Problem**
  - Limited set of buffers
P/C - Solution

Shared Full: semaphore \texttt{\leftarrow} 0;
Empty semaphore \texttt{\leftarrow} MaxBuffers;
MEPC: semaphore \texttt{\leftarrow} 1;

\begin{itemize}
\item Producer
\begin{verbatim}
Begin
  ...
P(Empty);
P(MEPC);
<add item to buffer>
V(MEPC);
V(Full);
  ...
End;
\end{verbatim}
\item Consumer
\begin{verbatim}
Begin
  ...
P(Full);
P(MEPC);
<remove item from buffer>
V(MEPC);
V(Empty);
  ...
End;
\end{verbatim}
\end{itemize}
P/C – Another Look

Pool full of Baskets

Consumer

Pool of empty Baskets

Producer
P/C – Another Look

- 9 Baskets – Bounded

- Consumer – Empties basket
  - Can only remove basket from Full Pool, if one is there
  - => Need “full” count
  - Emptys basket and places it in Empty pool

- Producer – Fills basket
  - Can only remove basket from Empty pool, if one is there
  - => Need “empty” count
  - Fills basket and places it in Full pool
P/C - Another Look

Shared semaphore: Emutex = 1, Fmutex = 1; full = 0, empty = 9;
Shared buf_type: buffer[9];

producer() {
    buf_type *next, *here;
    while(True) {
        produce_item(next);
P(empty); /*Claim empty buffer*/
P(Emutex); /*Manipulate the pool*/
here = obtain(empty);
V(Emutex);
copy_buffer(next, here);
P(Fmutex); /*Manipulate the pool*/
release(here, fullpool);
V(Fmutex); /*Signal full buffer*/
V(full);
    }
}

consumer() {
    buf_type *next, *here;
    while(True) {
        P(full); /*Claim full buffer*/
P(Fmutex); /*Manipulate the pool*/
here = obtain(full);
V(Fmutex);
copy_buffer(here, next);
P(Emutex); /*Manipulate the pool*/
release(here, emptypool);
V(Emutex); /*Signal empty buffer*/
V(empty);
consume_item(next);
    }
}
P/C - Example

- How realistic is PCP scenario?
- Consider a circular buffer
  - 12 slots
  - Producer points at next one it will fill
  - Consumer points at next one it will empty
- Don’t want:
  Producer = Consumer
  => (1) Consumer “consumed” faster than producer “produced”, or
  (2) Producer “produced” faster than consumer “consumed”.

Do we need to synchronize access to buffer?
P/C – Real World Scenario

- CPU can produce data faster than terminal can accept or viewer can read

```
Communication buffers in both
Xon/Xoff Flow Control
```
Readers / Writers Problem (Classic)

- Multiple readers of the same file?
  - No problem
- Multiple writers to the same file?
  - Might be a problem writing same record
    => Potentially a “lost update”
- Writing while reading
  - Might be a problem – read might occur while being written
    => Inconsistent data

Readers – Writers Problem

- Critical resource
  - File

- Consider multiple processes which can read or write to the file

- What constraints must be placed on these processes?
  - Many readers may read at one time
  - Mutual exclusion between readers and writers
  - Mutual exclusion between writers
Strong Reader Solution

Shared int: readCount = 0;
    semaphore: mutexRC = 1, writeBlock = 1;

reader(){
    while(TRUE) {
        P(mutexRC);
        readCount = readCount + 1;
        if (readCount == 1)
            P(writeBlock);
        V(mutexRC);
        access_file;
        P(mutexRC);
        readCount = readCount - 1;
        if (readCount == 0)
            V(writeBlock);
        V(mutexRC);
    }
}

writer(){
    while(TRUE) {
        P(writeBlock);
        access_file;
        V(writeBlock);
    }
}

This solution gives preference to Readers

If a reader has access to file and other readers want access, they get it... all writers must wait until all readers are done
Reader / Writers – Ver 2

- Create a Strong Writer

- Give priority to a waiting writer

- If a writer wishes to access the file, then it must be the next process to enter its critical section
Strong Writers Solution

Shared int: readCount = 0, writeCount = 0
  semaphore: mutex1 = 1, mutex2 = 1, readBlock = 1, writePending = 1, writeBlock = 1;

reader()
{
  while(TRUE) {
    P(writePending);
    P(readBlock);
    P(mutex1);
    readCount = readCount + 1;
    if (readCount == 1) then
      P(writeBlock);
    V(mutex1);
    V(readBlock);
    V(writePending);
    access file;
    P(mutex1);
    readCount = readCount - 1;
    if (readCount == 0) then
      V(writeBlock);
    V(mutex1);
  }
}

writer()
{
  while(TRUE) {
    P(mutex2);
    writeCount = writeCount + 1;
    if (writeCount == 1) then
      P(readBlock);
    V(mutex2);
    P(writeBlock);
    access file;
    V(writeBlock);
    P(mutex2);
    writeCount = writeCount - 1;
    if (writeCount == 0) then
      V(readBlock);
    V(mutex2);
  }
}
Implementing Counting Semaphores

```c
struct semaphore {
    int value = <initial value>;
    boolean mutex = FALSE;
    boolean hold = TRUE;
};

Shared struct semaphore s;

P(struct semaphore s) {
    while( TS(s.mutex) );
    s.value = s.value - 1;
    if (s.value < 0) {
        s.mutex = FALSE;
        while( TS(s.hold) );
    }
    else {
        s.mutex = FALSE;
    }
}

V(struct semaphore s) {
    while( TS(s.mutex) );
    s.value = s.value + 1;
    if (s.value <= 0) {
        while( !s.hold );
        s.hold = FALSE;
    }
    s.mutex = FALSE;
}
```