Chapter 8

Basic Synchronization Principles

Need for Synchronization

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

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

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

- Clearly, synchronization is needed if
  - A wants B to read x after it writes it & before it re-writes

Critical Section Problem

shared float balance;

```c
/* Code schema for p1 */
/* Code schema for p2 */
load R1, balance
load R2, amount
add R1, R2
store R1, balance
load R1, balance
load R2, amount
sub R1, R2
store R1, balance
```

Critical Section Problem...

```c
/* Schema for p1 */
/* Schema for p2 */
load R1, balance
load R2, amount
add R1, R2
store R1, balance

load R1, balance
load R2, amount
sub R1, R2
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 the same time

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

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

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

Shared integer: processnumber = 1;

Single global variable force lockstep synchronization

---

Using Shared Global Variables – Ver 2

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

void procestwo();
{
    while (true)
        {     // Hardwait
            while (processnumber == 1)
                ;
            criticalsectiontwo;
            processnumber = 1;
            otherstufftwo;
        }
}
```

Shared boolean: plinside = false, plinside = false;

• Process 1 & 2 can both be in the critical sections at the same time
  Because Test & Set operations are not atomic
  => Move setting of plinside/p2inside before test

---

Using Shared Global Variables – Ver 3

```
void processone();
{
    while (true) {
        plwantsin = true;
        while (p2wantsin) ;
        criticalsectionone;
        plwantsin = false;
        otherstuffone;
    }
}

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

Shared boolean: plwantsin = false, p2wantsin = false;

Deadlock can occur if both sets flag at the same time

=> Need a way to break out of loops....

---

Using Shared Global Variables – Peterson

```
void processone();
{
    while (true) {
        plwantsin = true;
        while (p2wantsin)
            ;
        criticalsectionone;
        plwantsin = false;
        otherstuffone;
    }
}

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

Shared boolean: plwantsin = false, p2wantsin = false;

Shared int: will_wait;

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

```
shared double balance;

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

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

---

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;

shared boolean lock = FALSE;
shared float balance;

// 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

enter(lock) {

disableInterrupts();

/* Loop until lock TRUE */
while (lock) {

    /* Let interrupts occur */
    enableInterrupts();

    disableInterrupts();

    lock = TRUE;

    enableInterrupts();
}

exit(lock) {

disableInterrupts();

/* Loop until lock FALSE */
while (lock) {

    /* Let interrupts occur */
    enableInterrupts();

    disableInterrupts();

    lock = FALSE;

    enableInterrupts();
}

// If 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 LnthLK = False;

/* Program for P1 */

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

/* 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 of information
- But try to minimize time component locked out

Protecting Multiple Components: 1st try

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

/* Program for P1 */

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

/* Program for P2 */

Suppose: P1... ; P2 runs & finishes; P1 ... ......
Any access to lngth 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);
    <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 requests and gets R1
        P2 requests and gets R2

P1 requests R2 and blocks
P2 requests R1 and blocks

⇒ DEADLOCK

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 3rd and much more preferable method
  - 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

- 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

\texttt{var x : semaphore}

Semaphore: Example

\begin{tabular}{|c|c|}
\hline
\textbf{Critical resource} & T \\
\hline
\textbf{Semaphore} & \textbf{\texttt{s = initial\_value}} \\
\hline
\textbf{Processes} & \textbf{A,B} \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
\textbf{Process A} & \textbf{Process B} \\
\hline
\texttt{P(S);} & \texttt{P(S);} \\
\hline
\texttt{<CS> /\* access T */} & \texttt{<CS> /\* access T */} \\
\hline
\texttt{V(S);} & \texttt{V(S);} \\
\hline
\end{tabular}
Semaphore: Example...

```
var S : semaphore ← 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>P(S);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;CS&gt;</td>
<td>P(S);</td>
<td></td>
</tr>
<tr>
<td>V(S);</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></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

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

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

Shared semaphore mutex = 1;

Using Semaphores - Example 1

```
proc_D() {
  while(true) {
    P(mutex);
    balance = balance + amount;
    V(mutex);
    ...
  }
}

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

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

```
proc_A() {
  while(true) {
    <compute A1>
    write(x);
    V(s1);
    <compute B1>
    P(s1);
    read(y);
    <compute B2>
    write(x);
    V(s2);
  }
}
```

```
proc_B() {
  while(true) {
    P(s1);
    read(x);
    <compute B1>
    write(y);
    V(s2);
    <compute A1>
    write(x);
  }
}
```

```
semaphore s = 1;
```

```
while( TS(s) ) {
  <critical section>
  S = FALSE;
  ...
  V(s);
  ...
}
```

```
boolean s = FALSE;
```

```
while( TS(s) ) {
  P(s);
  <critical section>
  S = TRUE;
  ...
  V(s);
  ...
}
```

```
TS(s) {
  Test s
  Set s to True
  Return original value
}
```

Note: No actual queueing, each process just "hard waits"

Using Hardware Test & Set [TS(s)] to Implement Binary Semaphore “Semantics”

- 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

CS 3204 - Arthur 25

CS 3204 - Arthur 26

CS 3204 - Arthur 27

CS 3204 - Arthur 28

CS 3204 - Arthur 29

CS 3204 - Arthur 30
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

```
begin
  ...
  P(Empty);
  P(MEPC);
  <add item to buffer>
  V(MEPC);
  V(Full);
  ...

End;

begin
  ...
  P(Full);
  P(MEPC);
  <remove item from buffer>
  V(MEPC);
  V(Empty);
  ...

End;
```

P/C – Another Look

Diagram showing consumer and producer accessing a pool of buffers.
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

```
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);
    }
}
```

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

---

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.”

---

P/C - Real World Scenario

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

```
CPU                 Terminal
```

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

```
file
```

---

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

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

```
writer() {
  while(TRUE) {
    access_file;
    P(mutexRC);
    writeCount = writeCount + 1;
    if (writeCount == 1) P(readBlock);
    V(writeBlock);
    lock_file;
    P(mutexRC);
    writeCount = writeCount - 1;
    if (writeCount == 0) V(readBlock);
    V(mutexRC);
  }
}
```

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

Solution gives preference to Readers

If a reader has access to file and other readers want access, they get it... all readers 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

```
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);
    lock_file;
    V(writeBlock);
    P(mutex2);
    writeCount = writeCount - 1;
    if (writeCount == 0) then V(readBlock);
    V(mutex2);
  }
}
```

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

Implementing Counting Semaphores

```
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( !s.hold );
  }
  else {
    s.mutex = FALSE;
  }
}
```

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