Chapter 7: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris & Windows

Background

- Concurrent access to shared data may result in data inconsistency
- Need mechanisms to ensure the orderly execution of cooperating processes

Illustration

- bounded-buffer problem (Chapter 4)
  - allows at most \((n-1)\) items in buffer at the same time
  - can we use \(n\) items?
  - suppose that we modify the producer-consumer code by adding a variable counter, initialized to 0 and incremented each time a new item is added to the buffer
Bounded-Buffer

Shared data:

```c
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Producer process:

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

Consumer process:

```c
item nextConsumed;
while (1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```
The statements
- \texttt{counter++;}
- \texttt{counter--};

must be performed \textit{atomically}

Atomic operation completes in its entirety without interruption

---

statement "\texttt{count++}" as implemented in machine language:

- \texttt{register1 = counter}
- \texttt{register1 = register1 + 1}
- \texttt{counter = register1}

statement "\texttt{count--}" may be implemented as:

- \texttt{register2 = counter}
- \texttt{register2 = register2 - 1}
- \texttt{counter = register2}

Interleaving depends upon how the producer and consumer processes are scheduled

---

Assume \texttt{counter} is initially 5.

One interleaving of statements is:

- producer: \texttt{register1 = counter (register1 = 5)}
- producer: \texttt{register1 = register1 + 1 (register1 = 6)}
- consumer: \texttt{register2 = counter (register2 = 5)}
- consumer: \texttt{register2 = register2 - 1 (register2 = 4)}
- consumer: \texttt{counter = register1 (counter = 6)}
- consumer: \texttt{counter = register2 (counter = 4)}

\texttt{counter} may be either 4 or 6, correct value is 5
Bounded Buffer: Race Condition

- several processes access and manipulate shared data concurrently
- The final value of the shared data depends upon which process finishes last
- To prevent race conditions, concurrent processes must be synchronized

The Critical-Section Problem

- n processes compete to use some shared data
- Each process has a code segment critical section in which the shared data is accessed
- Problem: ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section

Solution to Critical-Section Problem

1. Mutual Exclusion
   - If process Pi 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
Initial Attempts to Solve Problem

- Only 2 processes, \( P_0 \) and \( P_1 \).
- General structure of process \( P_i \) (other process \( P_j \))
  
  ```
  do {
    entry section
    critical section
    exit section
    remainder section
  } while (1);
  ```

- Idea: processes share common variables to synchronize their actions

---

Algorithm 1

- Shared variables:
  - int turn; initially turn = 0
  - turn = i \( \Rightarrow P_i \) can enter its critical section

- Process \( P_i \)
  
  ```
  do {
    while (turn != i);:
      critical section
      turn = j;
      remainder section
  } while (1);
  ```

Satisfies mutual exclusion
Requires strict alternation, violates progress req.

---

Algorithm 2

- Shared variables
  - boolean flag[2];
  - initially flag[0] = flag[1] = false.
  - flag[i] = true \( \Rightarrow P_i \) ready to enter its critical section

- Process \( P_i \)
  
  ```
  do {
    flag[i] := true;
    while (flag[i]) : Race
      critical section
      flag [i] = false;
      remainder section
  } while (1);
  ```

Satisfies mutual exclusion
Violates progress requirement
Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process $P_i$
  
  ```
  do {
    flag[i] := true;
    turn = j;
    while (flag[j] and turn = j) ;
    critical section
    flag[i] = false;
  } while (1);
  ```

- Meets all three requirements
  - solves the critical-section problem for two processes

Bakery Algorithm

Critical section for $n$ processes

- Before entering its critical section
  - process receives a number
- Holder of the smallest number enters the critical section
- If processes $P_i$ and $P_j$ receive the same number,
  - if $i < j$, then $P_i$ is served first; else $P_j$ is served first
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., $1,2,3,3,3,3,4,5...$

Bakery Algorithm

- Notation $< \equiv$ lexicographical order (ticket #, process id #)
  - $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - max $(a_0, ..., a_n)$ is a number, $k$, such that $k \geq a$ for $i = 0, ..., n - 1$
- Shared data
  - boolean choosing[n];
  - int number[n];
  - Data structures are initialized to false and 0 respectively
Bakery Algorithm

```c
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
        while (choosing[j]) ;
        while ((number[j] != 0) && (number[j],j) < number[i],i)) ;
    }
    critical section
    number[i] = 0;
    remainder section
} while (1);
```

Synchronization via 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;

// Process P
    do {
        while (!TestAndSet(lock)) ;
        critical section
        lock = false;
        remainder section
    }
```
Synchronization Hardware

<table>
<thead>
<tr>
<th>Atomically swap two variables.</th>
</tr>
</thead>
<tbody>
<tr>
<td>void Swap(boolean &amp;a, boolean &amp;b) {</td>
</tr>
<tr>
<td>boolean temp = a;</td>
</tr>
<tr>
<td>a = b;</td>
</tr>
<tr>
<td>b = temp;</td>
</tr>
</tbody>
</table>
| }

Mutual Exclusion with Swap

<table>
<thead>
<tr>
<th>Shared data (initialized to false):</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean lock;</td>
</tr>
<tr>
<td>boolean waiting[n];</td>
</tr>
<tr>
<td>Process P</td>
</tr>
<tr>
<td>do {</td>
</tr>
<tr>
<td>key = true;</td>
</tr>
<tr>
<td>while (key == true)</td>
</tr>
<tr>
<td>Swap(lock, key);</td>
</tr>
<tr>
<td>critical section</td>
</tr>
<tr>
<td>lock = false;</td>
</tr>
<tr>
<td>remainder section</td>
</tr>
</tbody>
</table>
| }

Semaphores

<table>
<thead>
<tr>
<th>Synchronization tool that does not require busy waiting</th>
</tr>
</thead>
<tbody>
<tr>
<td>can only be accessed via two atomic operations</td>
</tr>
<tr>
<td>wait(S)</td>
</tr>
<tr>
<td>wait while S &lt; 0</td>
</tr>
<tr>
<td>decrement S</td>
</tr>
<tr>
<td>signal(S)</td>
</tr>
<tr>
<td>increment S</td>
</tr>
<tr>
<td>Think of S as integer number</td>
</tr>
</tbody>
</table>
Critical Section of \( n \) Processes

- Shared data:
  - semaphore \( S \); initially \( S = 1 \)
- Process \( P_i \):
  
  ```
  do {
    wait(S);
    critical section
    signal(S);
    remainder section
  } while (1);
  ```

Semaphore Implementation

- Define a semaphore as a record
  ```
  typedef struct {
    int value;
    struct process *L;
  } semaphore;
  ```
- Assume two simple operations:
  - `block` suspends the process that invokes it
  - `wakeup(P)` resumes the execution of a blocked process \( P \)

Implementation

- Semaphore operations now defined as
  ```
  - wait(S):
    S.value--;
    if (S.value < 0) {
      add this process to S.L;
      block;
    }
  
  - signal(S):
    S.value++;
    if (S.value <= 0) {
      remove a process P from S.L;
      wakeup(P);
    }
  ```
Semaphore as a General Synchronization Tool

- Consider statement A and B in 2 processes
- Goal: execute B in P₂ only after A executed in P₁
- Use semaphore flag initialized to 0
- Code:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$P_2$</td>
</tr>
<tr>
<td>A</td>
<td>wait(flag)</td>
</tr>
<tr>
<td>signal(flag)</td>
<td>B</td>
</tr>
</tbody>
</table>

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
- Let S and Q be two semaphores initialized to 1

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>P₁</td>
</tr>
<tr>
<td>wait(S);</td>
<td>wait(Q);</td>
</tr>
<tr>
<td>wait(Q);</td>
<td>wait(S);</td>
</tr>
<tr>
<td>signal(S);</td>
<td>signal(Q);</td>
</tr>
<tr>
<td>signal(Q)</td>
<td>signal(S);</td>
</tr>
</tbody>
</table>

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

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
  - simpler to implement.
- Can implement a counting semaphore S as a binary semaphore.
Implementing $S$ as a Binary Semaphore

- Data structures:
  - binary-semaphore $S_1, S_2$
  - int $C$

- Initialization:
  - $S_1 = 1$
  - $S_2 = 0$
  - $C = \text{initial value of semaphore } S$

Implementing $S$

- wait operation
  - $\text{wait}(S_1);$
  - $C--;$
  - if ($C < 0$) {
    - $\text{signal}(S_1);$  
    - $\text{wait}(S_2);$ 
    - $\text{signal}(S_1);$ 
  } 

- signal operation
  - $\text{wait}(S_1);$ 
  - $C++;$
  - if ($C <= 0$) $\text{signal}(S_2);$ 
  - else 
    - $\text{signal}(S_1);$ 

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
Bounded-Buffer Problem

- Shared data
  - semaphore full, empty, mutex;

Initially:
full = 0, empty = n, mutex = 1

Bounded-Buffer Problem Producer Process

do {
  produce an item in nextp
  wait(empty);
  wait(mutex);
  add nextp to buffer
  signal(mutex);
  signal(full);
} while (1);

Bounded-Buffer Problem Consumer Process

do {
  wait(full)
  wait(mutex);
  remove an item from buffer to nextc
  signal(mutex);
  signal(empty);
  consume the item in nextc
} while (1);
Readers-Writers Problem

- Shared data
  - semaphore mutex, wrt;
- Initially
  - mutex = 1, wrt = 1, readcount = 0

Readers-Writers Problem Writer Process

- \texttt{wait(wrt)};
  - ...
  - writing is performed
  - ...
  - \texttt{signal(wrt)};

Readers-Writers Problem Reader Process

- \texttt{wait(mutex)};
  - readcount++;
  - if (readcount = 1)
    - \texttt{wait(wrt)};
    - \texttt{signal(mutex)};
  - reading is performed
  - ...
- \texttt{wait(mutex)};
  - readcount--;
  - if (readcount = 0)
    - \texttt{signal(wrt)};
    - \texttt{signal(mutex)}:
Dining-Philosophers Problem

- Shared data
  semaphore chopstick[5];
  Initially all values are 1

Philosopher $i$:

```c
    do {
        wait(chopstick[i])
        wait(chopstick[(i+1) % 5])
        ...  
        eat
        ...  
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
        ...  
        think
    } while (1);
```

Deadlock?

Critical Regions

- High-level synchronization construct
- Variable $v$ accessed only inside statement $S$
  region $v$ when $B$ do $S$
- While statement $S$ is being executed, no other process can access variable $v$
- Statement $S$ is executed only if:
  - boolean expression $B$ is true
  - no other process is in a region associated with $v$
Example – Bounded Buffer

- Shared data:

```c
struct buffer {
    int pool[n];
    int count, in, out;
}
```

Bounded Buffer Producer Process

- Producer process inserts `nextp` into the shared buffer

```c
region buffer when (count < n) {
    pool[in] = nextp;
    in = (in+1) % n;
    count++;
}
```

Bounded Buffer Consumer Process

- Consumer process removes an item from the shared buffer and puts it in `nextc`

```c
region buffer when (count > 0) {
    nextc = pool[out];
    out = (out+1) % n;
    count--;
}
```
Implementation region $x$ when $B$ do $S$

- Associate with the shared variable $x$, the following variables:
  - semaphore mutex, first-delay, second-delay;
  - int first-count, second-count;
- Mutually exclusive access to the critical section is provided by mutex.
- If a process cannot enter the critical section because the Boolean expression $B$ is false, it initially waits on the first-delay semaphore; moved to the second-delay semaphore before it is allowed to reevaluate $B$.

Implementation (cont.)

- Keep track of the number of processes waiting on first-delay and second-delay, with first-count and second-count respectively.
- The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore.
- For an arbitrary queuing discipline, a more complicated implementation is required.

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
monitor monitor-name {
    shared variable declarations
    procedure body $P_1(...)$ {
        ...
    }
    procedure body $P_2(...)$ {
        ...
    }
    procedure body $P_n(...)$ {
        ...
    }
    { initialization code }
}
```
Monitors

- To allow a process to wait within the monitor, a condition variable must be declared, as
  condition x, y;
- Condition variable can only be used with the operations wait and signal.
  - x.wait();
    means that the process invoking this operation is suspended until another process invokes
  - x.signal();
    resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

Monitor With Condition Variables

Dining Philosophers Example

monitor dp {
  enum {thinking, hungry, eating} state[5];
  condition self[5];
  void pickup(int i) // following slides
  void putdown(int i) // following slides
  void test(int i) // following slides
  void init() {
    for (int i = 0; i < 5; i++)
      state[i] = thinking;
  }
}
Dining Philosophers

```c
void pickup(int i) {
    state[i] = hungry;
    test[i];
    if (state[i] != eating)
        self[i].wait();
}

void putdown(int i) {
    state[i] = thinking;
    // test left and right neighbors
    test[(i+4) % 5];
    test[(i+1) % 5];
}
```

Dining Philosophers

```c
void test(int i) {
    if ( (state[(i + 4) % 5] != eating) &&
        (state[i] == hungry) &&
        (state[(i + 1) % 5] != eating)) {
        state[i] = eating;
        self[i].signal();
    }
}
```

Monitor Implementation Using Semaphores

- **Variables**
  - semaphore mutex; // (initially = 1)
  - semaphore next; // (initially = 0)
  - int next-count = 0;
- Each external procedure \( F \) will be replaced by
  ```c
  wait(mutex);
  ... 
  body of \( F \);
  ...
  if (next-count > 0)
      signal(next)
  else
      signal(mutex);
  ```
- Mutual exclusion within a monitor is ensured.
For each condition variable \( x \), we have:

\[
\text{semaphore } x\text{-sem}; \quad (\text{initially } = 0)
\]
\[
\text{int } x\text{-count } = 0;
\]

The operation \( x\text{-wait} \) can be implemented as:

\[
\begin{align*}
\text{x-count} & \text{++;} \\
\text{if( next-count } & > 0) \\
\text{signal(next);} \\
\text{else} \\
\text{signal(mutex);} \\
\text{wait(x-sem);} \\
\text{x-count} & \text{--;}
\end{align*}
\]

The operation \( x\text{-signal} \) can be implemented as:

\[
\begin{align*}
\text{if( x-count } & > 0) \\
\text{next-count} & \text{++;} \\
\text{signal(x-sem);} \\
\text{wait(next);} \\
\text{next-count} & \text{--;}
\end{align*}
\]

Conditional-wait construct: \( x\text{-wait}(c) \):

- \( c \) – integer read for each \( wait \) operation
- \( c \) (a priority number) stored with the waiting process
- when \( x\text{-signal} \) is executed, process with smallest associated priority number is resumed next

Check two conditions to establish correctness of system:

- User processes issue monitor calls in a correct sequence
- No process may ignore the mutual-exclusion gateway provided by the monitor, or try to access the shared resource directly, without using the access protocols.
Solaris 2 Synchronization

- Implements a variety of locks to support
  - multitasking
  - multithreading (including real-time threads)
  - multiprocessing.
- adaptive mutexes
  - efficient protection of data in short code segments
- condition variables and readers-writers
  - locks when longer sections of code need access to data
- Turnstiles
  - order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows 2000 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 either mutexes or semaphores
- Dispatcher objects may also provide events
  - event acts much like a condition variable