Chapter 6
Process Synchronization

张竞慧
办公室：计算机楼366室
电邮：jhzhang@seu.edu.cn
主页：http://cse.seu.edu.cn/PersonalPage/zjh/
电话：025-52091017
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples
Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem (Chapter 3) allows at most $n - 1$ items in buffer at the same time. A solution, where all $N$ buffers are used is not simple.

- Suppose that we modify the producer-consumer code by adding a variable *counter*
Producer-Consumer Problem (Review)

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.
Bounded-Buffer – Shared-Memory Solution (Review)

■ Shared data

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

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

Bounded-Buffer – Producer Process (Review)
Bounded-Buffer – Consumer Process (Review)

```c
item nextConsumed;

while (1) {
    while (in == out)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```
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;
```
Bounded-Buffer

Producer process

item nextProduced;

while (1) {
    while (counter == BUFFER_SIZE);
    /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
Bounded-Buffer

- Consumer process

```c
item nextConsumed;

while (1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```
Bounded Buffer

- The statements
  
  ```
  counter++;  
  counter--;  
  ```

  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.
Bounded Buffer

The statement “count++” may be implemented in machine language as:

\[
\text{register1} = \text{counter} \\
\text{register1} = \text{register1} + 1 \\
\text{counter} = \text{register1}
\]

The statement “count--” may be implemented as:

\[
\text{register2} = \text{counter} \\
\text{register2} = \text{register2} - 1 \\
\text{counter} = \text{register2}
\]
Bounded Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.
Bounded Buffer

Assume `counter` is initially 5. One interleaving of statements is:
producer: `register1 = counter` \((register1 = 5)\)
producer: `register1 = register1 + 1` \((register1 = 6)\)
consumer: `register2 = counter` \((register2 = 5)\)
consumer: `register2 = register2 – 1` \((register2 = 4)\)
producer: `counter = register1` \((counter = 6)\)
consumer: `counter = register2` \((counter = 4)\)

The value of `counter` may be either 4 or 6, where the correct result should be 5.
register1 = counter
register1 = register1 + 1
counter = register1

register2 = counter
register2 = register2 – 1
counter = register2
Race Condition

- **Race condition** occurs, if:
  - two or more processes/threads access and manipulate the same data concurrently, and
  - the outcome of the execution depends on the particular order in which the access takes place.

- To prevent race conditions, concurrent processes must be **synchronized**.
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples
The Critical-Section Problem

- $n$ processes all competing to use some shared data

- Each process has a code segment, called *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.
Thus, the execution of critical sections must be *mutually exclusive* (e.g., at most one process can be in its critical section at any time).

The *critical-section problem* is to design a protocol that processes can use to cooperate.
The Critical Section Protocol

A critical section protocol consists of two parts: an entry section and an exit section.

Between them is the critical section that must run in a mutually exclusive way.
Solution to Critical-Section Problem

Any solution to the critical section problem must satisfy the following three conditions:

- Mutual Exclusion
- Progress
- Bounded Waiting

Moreover, the solution cannot depend on relative speed of processes and scheduling policy.
Mutual Exclusion

- If a process $P$ is executing in its critical section, then \textit{no} other processes can be executing in their critical sections.

- The \textit{entry protocol} should be capable of blocking processes that wish to enter but cannot.

- Moreover, when the process that is executing in its critical section exits, the \textit{entry protocol} must be able to know this fact and allows a waiting process to enter.
Progress

- If no process is executing in its critical section and some processes wish to enter their critical sections, then
  
  - Only those processes that are waiting to enter can participate in the competition (to enter their critical sections).
  - No other process can influence this decision.
  - This decision cannot be postponed indefinitely.
Bounded Waiting

After a process made a request to enter its critical section and before it is granted the permission to enter, there exists a bound on the number of times that other processes are allowed to enter.

Hence, even though a process may be blocked by other waiting processes, it will not be waiting forever.

- 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 $P_j$)

```plaintext
do {
  entry section
  critical section
  exit section
  remainder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.
Algorithm 1

- Shared variables:
  - `int turn;`
  - Initially `turn = i` (or `turn = j`)

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

- are forced to run in an alternating way.

- Satisfies **mutual exclusion**, but not **progress**
Algorithm 2

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

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

- Satisfies mutual exclusion, but not progress requirement.
Consider the following trace:

\[P_0\text{ sets flag}[0]\text{ to true}
\]  
A context-switch occurs  
\[P_1\text{ sets flag}[1]\text{ to true}
\]  
\[P_1\text{ loops in while}
\]  
A context-switch occurs  
\[P_0\text{ loops in while}
\]

Both \(P_0\) and \(P_1\) loop forever. This is the livelock. No progress.
Is the following algorithm correct?

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

- **Process \( P_i \):**
  ```
  do {
    while (flag[j]) ;
    flag[i] = true;
    critical section
    flag [i] = false;
    remainder section
  } while (1);
  ```
Consider the following trace:

\[ P_0 \text{ finds } (\text{flag}[1] == \text{false}) \]
\[ \text{The scheduler forces a context-switch} \]
\[ P_1 \text{ (finds } \text{flag}[0] == \text{false}) \]
\[ P_1 \text{ sets } (\text{flag}[1] = \text{true}) \]
\[ P_1 \text{ enters the critical section} \]
\[ \text{The scheduler forces a context-switch} \]
\[ P_0 \text{ sets } (\text{flag}[0] = \text{true}) \]
\[ P_0 \text{ enters the critical section} \]

Both \( P_0 \) and \( P_1 \) are now in the critical section

With both processes in the critical section, the mutual exclusion criteria has been violated.
Algorithm 3

- Combined shared variables of algorithms 1, 2.
- Process $P_i$

```plaintext
do {
    flag [i] := true;
    turn = j;
    while (flag [j] and turn == j) ;
    critical section
    flag [i] = false;
    remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes.
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples
Hardware Support

- There are two types of hardware synchronization supports:
  - Disabling/Enabling interrupts: This is slow and difficult to implement on multiprocessor systems.
  - Special machine instructions:
    - Test and set (TS)
    - Swap
Interrupt Disabling

- Because interrupts are disabled, no context switch will occur in a critical section.
- Infeasible in a multiprocessor system because all CPUs must be informed.
- Some features that depend on interrupts (e.g., clock) may not work properly.
Test-and-Set

- Test and modify the content of a word atomically.

```c
boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
```
Mutual Exclusion with Test-and-Set

- Shared data:
  ```java
  boolean lock = false;
  ```

- Process $P_i$
  ```java
  do {
    while (TestAndSet(&lock)) ;
    critical section
    lock = false;
    remainder section
  }
  ```
Swap

- Atomically swap two variables.

```c
void Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```
Mutual Exclusion with Swap

- Shared data (initialized to \texttt{false}):  
  
  \begin{verbatim}
  boolean lock;
  \end{verbatim}

- Local variable
  
  \begin{verbatim}
  boolean key;
  \end{verbatim}

- Process $P_i$
  
  \begin{verbatim}
  do {
    key = true;
    while (key == true)
      Swap(lock,key);
    critical section
    lock = false;
  remainder section
  }
  \end{verbatim}
Bounded Waiting Mutual Exclusion with TestAndSet

Enter Critical Section

waiting[i] = true;
key = true;
while (waiting[i] && key)
    key = TestAndSet(lock);
waiting[i] = false;

Leave Critical Section

j = (i+1)\%n
while ((j!=i) && !waiting[j])
    j = (j+1)\%n;
if (j == i)
    lock = false;
else
    waiting[j] = false;
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples
Semaphores

- Semaphore $S$ – integer variable
- can only be accessed via two indivisible (atomic) operations

**wait** ($S$):

```while $S \leq 0$ do no-op;  
    $S$--;
```  

**signal** ($S$):

```$S$++;  ```
Critical Section of \( n \) Processes

- Shared data:
  
  \[
  \text{semaphore mutex}; \quad ///\text{initially mutex} = 1
  \]

- Process \( P_i \):

  \[
  \begin{align*}
  \text{do} \{ & \quad \text{critical section} \\
  & \quad \text{wait(mutex);} \\
  & \quad \text{signal(mutex);} \\
  \} \quad \text{while} \ (1);
  \end{align*}
  \]
Semaphore as a General Synchronization Tool

- Execute $B$ in $P_j$ only after $A$ executed in $P_i$
- Use semaphore flag initialized to 0
- Code:

\[
P_i \quad \quad \quad \quad P_j
\]
\[
\vdots \quad \quad \quad \quad \vdots
\]
\[
A \quad \quad \quad \quad \text{wait(flag)}
\]
\[
signal(flag) \quad \quad B
\]
Semaphore Implementation

- Define a semaphore as a record

  ```c
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.
method signal

method wait

semaphore

counter

waiting list
Semaphore operations now defined as

\textit{wait}(S):
\begin{verbatim}
    S.value--; \\
    if (S.value < 0) {
        add this process to S.L; \\
        block;
    }
\end{verbatim}

\textit{signal}(S):
\begin{verbatim}
    S.value++; \\
    if (S.value <= 0) {
        remove a process P from S.L; \\
        wakeup(P);
    }
\end{verbatim}
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_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.
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples
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

```c
semaphore mutex, wrt;
```

Initially

```
mutex = 1, wrt = 1, readcount = 0
```
Reader-Writers Problem

Process

wait(wrt);

... writing is performed ...

signal(wrt);
Readers-Writers Problem Reader Process

wait(mutex);
readcount++;
if (readcount == 1)
    wait(wrt);
signal(mutex);

…
reading is performed

…

wait(mutex);
readcount--;
if (readcount == 0)
signal(wrt);
signal(mutex):
Dining-Philosophers Problem

- Shared data

```
semaphore chopstick[5];
```

Initially all values are 1
Dining-Philosophers Problem

Philosopher $i$:

\[ \text{do } \{
    \text{wait(chopstick}[i])}
    \text{wait(chopstick}[(i+1) \% 5])
    \ldots
    \text{eat}
    \ldots
    \text{signal(chopstick}[i]);}
    \text{signal(chopstick}[(i+1) \% 5]);
    \ldots
    \text{think}
    \ldots
\} \text{while (1);} \]
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples
Monitors

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

```plaintext
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        . . .
    }
    procedure body P2 (...) {
        . . .
    }
    procedure body Pn (...) {
        . . .
    }
    { initialization code}
}
```
Monitors: Mutual Exclusion

- No more than one process can be executing within a monitor. Thus, mutual exclusion is guaranteed within a monitor.

- When a process calls a monitor procedure and enters the monitor successfully, it is the only process executing in the monitor.

- When a process calls a monitor procedure and the monitor has a process running, the caller will be blocked outside of the monitor.
Schematic View of a Monitor
processes waiting to enter monitor
Monitors

To allow a process to wait within the monitor, a **condition** variable must be declared, as

```java
condition x, y;
```

Condition variable can only be used with the operations **wait** and **signal**.

- The operation `x.wait();`
  means that the process invoking this operation is suspended until another process invokes `x.signal();`

- The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.
Consider the released process (from the signaled condition) and the process that signals. There are two processes executing in the monitor, and mutual exclusion is violated!

There are two common and popular approaches to address this problem:

- The released process takes the monitor and the signaling process waits somewhere.
- The released process waits somewhere and the signaling process continues to use the monitor.
# Semaphore vs. Condition

<table>
<thead>
<tr>
<th>Semaphores</th>
<th>Condition Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used anywhere, but not in a monitor</td>
<td>Can only be used in monitors</td>
</tr>
<tr>
<td><code>wait()</code> does not always block its caller</td>
<td><code>wait()</code> <strong>always</strong> blocks its caller</td>
</tr>
<tr>
<td><code>signal()</code> either releases a process, or increases the semaphore counter</td>
<td><code>signal()</code> either releases a process, or the signal is <strong>lost</strong> as if it never occurs</td>
</tr>
<tr>
<td>If <code>signal()</code> releases a process, the caller and the released <strong>both continue</strong></td>
<td>If <code>signal()</code> releases a process, either the caller or the released continues, but <strong>not both</strong></td>
</tr>
</tbody>
</table>
Monitor With Condition Variables

- Queues associated with $x, y$ conditions
- Shared data
- Operations
- Initialization code
- Entry queue
waiting to enter the monitor

entry queue

waiting bench

reentering threads

Monitor Procedure

Monitor Procedure

Monitor Procedure

Initialization

Monitor Procedure

c.v. waiting list

c.v.

c.v.
Bounded Buffer Solution Using Monitor

```c
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];       /* space for N items */
int nextin, nextout;   /* buffer pointers */
int count;             /* number of items in buffer */
cond notfull, notempty; /* condition variables for synchronization */

void append (char x)
{
    if (count == N) cwait(notfull); /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);              /* resume any waiting consumer */
}

void take (char x)
{
    if (count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                        /* one fewer item in buffer */
    csignal(notfull);              /* resume any waiting producer */
}

{   nextin = 0; nextout = 0; count = 0; /* buffer initially empty */
}
```
void producer()
{
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}

void consumer()
{
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}

void main()
{
    parbegin (producer, consumer);
}
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

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
  ```
  wait(mutex);
  ```

  ```
  ... body of $F$; ...
  ```

  ```
  if (next-count > 0)
  signal(next)
  else signal(mutex);
  ```

- Mutual exclusion within a monitor is ensured.
Monitor Implementation

- For each condition variable $x$, we have:
  semaphore $x$-sem; // (initially = 0)
  int $x$-count = 0;

- The operation $x$.wait can be implemented as:

  $x$-count++;
  if (next-count > 0)
    signal(next);
  else
    signal(mutex);
  wait($x$-sem);
  $x$-count--;
Monitor Implementation

The operation `x.signal` can be implemented as:

```c
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}
```
Monitor Implementation

- **Conditional-wait** construct: `x.wait(c);`
  - `c` – integer expression evaluated when the `wait` operation is executed.
  - Value of `c` (a *priority number*) stored with the name of the process that is suspended.
  - When `x.signal` is executed, process with smallest associated priority number is resumed next.
Check two conditions to establish correctness of system:

- User processes must always make their calls on the monitor in a correct sequence.
- Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples
Solaris 2 Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Uses *adaptive mutexes* for efficiency when protecting data from short code segments.
- Uses *condition variables*, *semaphore*, and *readers-writers locks* when longer sections of code need access to data.
- Uses *turnstile* to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.
Windows XP Synchronization

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