Chapter 7: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
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-butter problem (Chapter 4) 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 $counter$, 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;
```
Bounded-Buffer

- Producer process

```c
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:

  register1 = counter
  register1 = register1 + 1
  counter = register1

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

  register2 = counter
  register2 = register2 – 1
  counter = 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.
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 **count** may be either 4 or 6, where the correct result should be 5.
Race Condition

- **Race condition**: The situation where 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 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.
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.
Initial Attempts to Solve Problem

- Only 2 processes, \( P_0 \) and \( P_1 \)
- General structure of process \( P_i \) (other process \( P_j \))
  
  \[
  \text{do} \{ \\
  \quad \text{entry section} \\
  \quad \text{critical section} \\
  \quad \text{exit section} \\
  \quad \text{reminder section} \\
  \} \text{ while (1);} \\
  \]

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

- Shared variables:
  - `int turn;`
  - Initially `turn = 0`
  - `turn - i` ⇒ `P_i` can enter its critical section

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

- 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.
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;
    remainder section
  } while (1);
  ```

- Meets all three requirements; solves the critical-section problem for two processes.
Synchronization Hardware

- Any solution to the critical section problem requires a simple tool – a lock.
- A process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.
- Modern computer systems provide special hardware to swap the contents of two words atomically, that is as one uninterruptible unit.
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 `false`):
  ```
  boolean lock;
  boolean waiting[n];
  ```

- Process $P_i$
  ```
  do {
    key = true;
    while (key == true)
      Swap(lock,key);
      critical section
      lock = false;
      remainder section
  }
  ```
Semaphores

- A synchronization tool that uses a variable that is used to control access to a common resource by multiple processes in a multiprogramming OS.
- Semaphore $S$ – integer variable
- can only be accessed via two indivisible (atomic) operations

```plaintext
wait (S):
    while $S \leq 0$ do no-op;
    $S--$;

signal (S):
    $S++$;
```
Semaphores

- The value of a **counting semaphore** can range over an unrestricted domain.
- The value of a **binary semaphore** can be 0 or 1.
- Binary semaphores are sometimes known as **mutex locks**, as they are locks that provide *mutual* exclusion.
Semaphores

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

- **Process** \(P_i\):

  ```
  \begin{array}{l}
  \text{do} \{ \\
  \quad \text{wait(\textit{mutex});} \\
  \quad \text{critical section} \\
  \quad \text{signal(\textit{mutex});} \\
  \quad \text{remainder section} \\
  \} \text{ while (1);}
  \end{array}
  ```
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`. 
Semaphore operations now defined as

\[ \text{wait}(S): \]
\[
S.\text{value}--; \\
\text{if (S.\text{value} < 0)} \{ \\
\quad \text{add this process to } S.L; \\
\quad \text{block}; \\
\} \\
\]

\[ \text{signal}(S): \]
\[
S.\text{value}++; \\
\text{if (S.\text{value} <= 0)} \{ \\
\quad \text{remove a process } P \text{ from } S.L; \\
\quad \text{wakeup}(P); \\
\} \]
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 \hspace{1cm} P_1
  \begin{align*}
  &\text{wait}(S); \hspace{1cm} \text{wait}(Q);
  
  &\text{wait}(Q); \hspace{1cm} \text{wait}(S);
  
  &\vdots \hspace{1cm} \vdots
  
  &\text{signal}(S); \hspace{1cm} \text{signal}(Q);
  
  &\text{signal}(Q) \hspace{1cm} \text{signal}(S);
  \end{align*}
  $$

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

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

- Shared data

`semaphore chopstick[5];`

Initially all values are 1
Dining-Philosophers Problem

Philosopher $i$:

\[
\text{do} \{ \\
\quad \text{wait(chopstick}[i])} \\
\quad \text{wait(chopstick}[i+1) \% 5])} \\
\quad \ldots \quad \text{eat} \quad \ldots \\
\quad \text{signal(chopstick}[i]);} \\
\quad \text{signal(chopstick}[i+1) \% 5]);} \\
\quad \ldots \quad \text{think} \quad \ldots \\
\} \text{ while (1);}
\]
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

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

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

    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 $x$.$\text{wait}$ can be implemented as:
  ```
  x-count++;
  if (next-count > 0)
    signal(next);
  else
    signal(mutex);
  wait(x-sem);
  x-count--;
  ```
The operation \texttt{x.signal} can be implemented as:

\begin{verbatim}
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}
\end{verbatim}