Chapter 7: Synchronization
Examples
7: Model Synchronization Problems

- The bounded-buffer, readers-writers, and dining philosophers synchronization problems.
- Tools used by Linux to solve synchronization problems.
- POSIX solutions to synchronization problems.
Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
Bounded-Buffer Problem

- $n$ buffers, each can hold one item
- Semaphore $\texttt{mutex}$ initialized to the value 1
- Semaphore $\texttt{full}$ initialized to the value 0
- Semaphore $\texttt{empty}$ initialized to the value $n$
The structure of the producer process

```c
while (true) {
    ... /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
}
```
Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```c
while (true) {
    wait(full);
    wait(mutex);
    ...
    /* remove an item from buffer to next_consumed */
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume the item in next consumed */
    ...
}
```
Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data set; they do **not** perform any updates
  - **Writers** – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities

Shared Data

- Data set
- Semaphore `rw_mutex` initialized to 1
- Semaphore `mutex` initialized to 1
- Integer `read_count` initialized to 0
Readers-Writers Problem (Cont.)

- The structure of a writer process

```java
while (true) {
    wait(rw_mutex);
    ...
    /* writing is performed */
    ...
    signal(rw_mutex);
}
```
The structure of a reader process

```c
while (true){
  wait(mutex);
  read_count++;
  if (read_count == 1)
    wait(rw_mutex);
  wait(rw_mutex);
  signal(mutex);
  ...
  /* reading is performed */
  ...
  wait(mutex);
  read_count--;
  if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
  signal(mutex);
}
```
Readers-Writers Problem Variations

- **First** variation – no reader kept waiting unless writer has permission to use shared object
- **Second** variation – once writer is ready, it writes as soon as any existing writer finishes writing
- Both may have starvation leading to even more variations
Dining Philosophers Problem

- Originally posed by Edsger Dijkstra in 1965 as a tape drive exercise for his students, and later formalized by C.A.R. Hoare.
- Five philosophers sit at a round table, and spend their lives alternating thinking and eating.
- They each have a bowl of spaghetti in front of them, and five forks are between the five bowls.
- They need two forks to eat. They cannot eat with just one.
- Problem has morphed over the years to bowls of rice and chopsticks.
Dining Philosophers Problem

- Philosophers are independent – they do not interact with their neighbors. They try to pick up 2 chopsticks one after the other to eat from bowl
  - Need both chopsticks to eat, then release both when done
- Shared data:
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1
Dining Philosophers Problem

- Formal Solution Requirements:
  - Only one philosopher can hold a chopstick at a time.
  - It must be deadlock-free
  - It must be impossible for a philosopher to starve waiting for a chopstick.
  - It must be possible for more than one philosopher to eat at the same time.
Dining Philosophers Algorithm

- Solution using semaphores
- The structure of Philosopher $i$:
  ```java
  while (true) {
      wait (chopstick[i] );
      wait (chopStick[ (i + 1) % 5 ] );

      /* eat for awhile */
      signal (chopstick[i] );
      signal (chopstick[ (i + 1) % 5 ] );

      /* think for awhile */
  }
  
  What is the problem with this algorithm?
Dining Philosophers #2

- A deadlock free solution uses an array of state variables and an array of semaphores:

```c
sem_t mutex;
sem_t S[N];
int   state[N];
```

- A philosopher tries to take forks as follows

```c
void take_forks(int ph_num)
{
    sem_wait(&mutex);
    state[ph_num] = HUNGRY;
    try_to_eat(ph_num);
    signal(mutex);
    wait(S[ph_num]); // wait here if could not eat
    sleep(1);
}
```
Dining Philosophers #2

Before a philosopher picks up any chopsticks, she checks whether her neighbors are holding any with something like this:

```c
if (state[i] == HUNGRY && state[(i+1)%5] != EATING && state[(i+4)%5] != EATING)
{
  state[i] = EATING;
  // can eat!!
  signal(S[i]);
}
```

The putting down of forks:

```c
void put_forks(int i)
{
  wait(mutex);
  state[i] = THINKING;
  try_to_eat(((i+1)%5));
  try_to_eat(((i+4)%5));
  signal(mutex);
}
```
The code for trying to eat:

```c
void try_to_eat (int i) {
    if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        signal(S[i]) ;
    }
}

initialization_code() {
    for (int i = 0; i < 5; i++)
       state[i] = THINKING;
}
```
Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive

- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both

- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption
Synchronization in Linux

- There are several options for process and thread synchronization in Linux.
  - Atomic variables
    - `atomic_t` is the type for atomic integer
    - Consider the variables
      ```c
      atomic_t counter;
      int value;
      ```

<table>
<thead>
<tr>
<th>Atomic Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atomic_set(&amp;counter, 5);</code></td>
<td>counter = 5</td>
</tr>
<tr>
<td><code>atomic_add(10, &amp;counter);</code></td>
<td>counter = counter + 10</td>
</tr>
<tr>
<td><code>atomic_sub(4, &amp;counter);</code></td>
<td>counter = counter - 4</td>
</tr>
<tr>
<td><code>atomic_inc(&amp;counter);</code></td>
<td>counter = counter + 1</td>
</tr>
<tr>
<td><code>value = atomic_read(&amp;counter);</code></td>
<td>value = 12</td>
</tr>
</tbody>
</table>
POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used on UNIX, Linux, and macOS
POSIX Mutex Locks

- Creating and initializing the lock
  ```c
  #include <pthread.h>
  
  pthread_mutex_t mutex;
  
  /* create and initialize the mutex lock */
  pthread_mutex_init(&mutex, NULL);
  ```

- Acquiring and releasing the lock
  ```c
  /* acquire the mutex lock */
  pthread_mutex_lock(&mutex);
  
  /* critical section */
  
  /* release the mutex lock */
  pthread_mutex_unlock(&mutex);
  ```
POSIX Semaphores

- POSIX provides two versions – **named** and **unnamed**.
- Named semaphores can be used by unrelated processes, unnamed cannot.
POSIX Named Semaphores

- Creating an initializing the semaphore:

```c
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name `SEM`.

- Acquiring and releasing the semaphore:

```c
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```
POSIX Unnamed Semaphores

Creating an initializing the semaphore:

```c
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

Acquiring and releasing the semaphore:

```c
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```
POSIX Condition Variables

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor, POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```c
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_var, NULL);
```
POSIX Condition Variables

- Thread waiting for the condition \( a == b \) to become true:

  ```c
  pthread_mutex_lock(&mutex);
  while (a != b)
  
      pthread_cond_wait(&cond_var, &mutex);
  
  pthread_mutex_unlock(&mutex);
  ```

- Thread signaling another thread waiting on the condition variable:

  ```c
  pthread_mutex_lock(&mutex);
  a = b;
  
  pthread_cond_signal(&cond_var);
  pthread_mutex_unlock(&mutex);
  ```
Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages
Consider a function update() that must be called atomically. One option is to use mutex locks:

```c
void update ()
{
    acquire();

    /* modify shared data */

    release();
}
```

A **memory transaction** is a sequence of read-write operations to memory that are performed atomically. A transaction can be completed by adding `atomic{S}` which ensures statements in $S$ are executed atomically:

```c
void update ()
{
    atomic {
        /* modify shared data */
    }
}
```
OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```c
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.
End of Chapter 7