Chapter 6: Process Synchronization

肖 卿 俊
办公室：九龙湖校区计算机楼212室
电邮：csqjxiao@seu.edu.cn
主页：https://csqjxiao.github.io/PersonalPage
电话：025-52091022
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Conditional Variables and Monitors
- Synchronization Examples
Concurrent access to shared data may result in data inconsistency.

Let us recall the concept of race condition:
- Several 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.

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
A Previously Used Example

```c
void *mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg)
    return NULL;
}
```

We split the for-loop across two threads running on two CPU cores.

void *mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg)
    return NULL;
}

```c
int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```
How much faster?

- We’re expecting a speedup of 2
- OK, perhaps a little less because of Amdahl’s Law (to predict the theoretical speedup when using multiple processors)
  - overhead for forking and joining multiple threads
- But it is actually slower!! Why??
- Here’s the mental picture that we have – two processors, shared memory
This mental picture is wrong!

We’ve forgotten about caches!

- The memory may be shared, but each processor has its own L1 cache
- As each processor updates *counter*, it bounces between L1 caches
The code is not only slow, it’s WRONG!

- Since the variable \textit{counter} is shared, we can get a data race.
- Increment operation: \textit{counter++}  
  
  MIPS equivalent:
  \begin{verbatim}
  lw $t0, counter
  addi $t0, $t0, 1
  sw $t0, counter
  \end{verbatim}

- A data race occurs when data is accessed and manipulated by multiple processors, and the outcome depends on the sequence or timing of these events.

<table>
<thead>
<tr>
<th>Sequence 1</th>
<th>Sequence 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor 1</td>
<td>Processor 2</td>
</tr>
<tr>
<td>lw $t0, counter</td>
<td>lw $t0, counter</td>
</tr>
<tr>
<td>addi $t0, $t0, 1</td>
<td>addi $t0, $t0, 1</td>
</tr>
<tr>
<td>sw $t0, counter</td>
<td>sw $t0, counter</td>
</tr>
</tbody>
</table>

\textit{counter} increases by 2  
\textit{counter} increases by 1 !!
Another Example: Revisit the Producer Consumer Problem

- Shared-memory solution to bounded-buffer problem (Chapter 3)
  - The code can only use N-1 items in the buffer

**Producer:**
```
while (1) {
    while (((in+1) % BUF_SIZE) == out) ;
    ......
in = (in+1) % BUF_SIZE;
}
```

**Consumer:**
```
while (1) {
    while (in == out) ;
    ......
    out = (out+1) % BUF_SIZE;
}
```

- We modify the above code by adding a variable *counter*, such that all items in the buffer can be used
Bounded-Buffer Solution

Shared data

```
#define BUF_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUF_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```
Bounded-Buffer Solution

**Producer process**

```c
item nextProduced;

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

**Consumer process**

```c
item nextConsumed;

while (1) {
    while (counter == 0) 
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUF_SIZE;
    counter--;
}
```
Critical Shared Data

- Counter is a piece of critical shared data.
- The statements

```java
counter++;
counter--;
```

must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.
Difficult to Implement the Atomic Guarantee

However, the statement “count++” may be implemented in machine language as:

\[
\begin{align*}
\text{register1} & = \text{counter} \\
\text{register1} & = \text{register1} + 1 \\
\text{counter} & = \text{register1}
\end{align*}
\]

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

\[
\begin{align*}
\text{register2} & = \text{counter} \\
\text{register2} & = \text{register2} - 1 \\
\text{counter} & = \text{register2}
\end{align*}
\]
Potential Data Inconsistency

- 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 \textit{counter} is initially 5. One interleaving of statements is:

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

The value of \textit{count} may be either 4 or 6, where the correct result should be 5.
Producer

\[\text{register}_1 = \text{counter}\]
\[\text{register}_1 = \text{register}_1 + 1\]
\[\text{counter} = \text{register}_1\]

Consumer

\[\text{register}_2 = \text{counter}\]
\[\text{register}_2 = \text{register}_2 - 1\]
\[\text{counter} = \text{register}_2\]
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
- Conditional Variables and Monitors
- Synchronization Examples
Three Typical Mechanisms of Process Synchronization

- Locks for shared memory programming
  - **Exclusive Lock**
  - **Shared Lock:**
    - Readers can share a lock, but readers and writers must access the data mutually exclusively.

- There are other synchronization primitives for shared memory programming, e.g., Barrier
OS Support to Implement an Exclusive Lock for Threads

- Using Mutex: Sleeping of threads Instead of spinning that wastes CPU cycles

- APIs of POSIX Thread for exclusive lock
  ```c
  int pthread_mutex_lock(pthread_mutex_t* mutex)
  int pthread_mutex_unlock(pthread_mutex_t* mutex)
  ```

- An Example
  ```c
  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
  pthread_mutex_lock(&lock);
  counter = counter+1; // or whatever your critical section is
  pthread_mutex_unlock(&lock);
  ```

Give a demonstration
The Critical-Section Problem: An Use Case of Exclusive Lock

- Multiple 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 (or lock) and an exit section (or unlock).

- 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 correctness of 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 *no* other processes can be executing in their critical sections.

- The *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 *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
Solve the Problem without any OS Support

- Consider a simple case of only 2 processes, $P_0$ and $P_1$
- General structure of process $P_i$ (and $P_j$)
  
  ```
  do {
      entry section
      critical section
      exit section
      remainder section
  } while (1);
  ```

- Processes may share some common variables to synchronize their actions.
Consider Three Test Cases

Case 1: One process repeatedly attempts to enter the critical section (CS)

◆ Progress Test: Whether $P_0$’s repeated entering of CS is independent of $P_1$’s attempt

Case 2: One process is already in the critical section, and meanwhile the other process attempts to enter

◆ Mutual Exclusive Test: $P_0$ safely block $P_1$ out

Case 3: Two processes try to enter the critical section simultaneously

◆ Progress Test: Whether it is possible for the two processes to block each other’s entry

◆ Mutual Exclusive Test: Whether it is possible for them to both enter the section
Our First Attempt: Algorithm 1

■ Shared variables:
  ◆ boolean lock;  // initially lock = false
  ◆ lock = true ⇒ the critical section has been locked

■ Process $P_i$:

  ```
  do {
    while (lock) ; // if locked then wait
    lock = true;  // acquire the lock
    critical section
    lock = false; // release the lock
    remainder section
  } while (1);
  ```

■ Does not satisfy mutual exclusion. Why?
Our Second Attempt: Algorithm 2

- Shared variables:
  - `int victim`; initially `victim = 0` (or `victim = 1`)
- Process $P_i$:
  ```
  do {victim = i; // determine who is the victim
       while (victim == i) ; // if I am victim, then wait
       critical section // assume empty
       // do nothing for CS exit
       remainder section}
  while (1);
  ```
- Processes are forced to run in an alternating way
- Satisfies mutual exclusion, but not progress
Alternating and Atomic Execution of Algorithm 2

thread 0

victim = 0
while(victim == 0);

wait

victim = 0
while(victim == 0);

wait

thread 1

victim = 1
while(victim == 1);

wait
Deadlock of Algorithm 2

```c
thread 0
victim = 0
while(victim == 0);
```

wait

deadlock!
Another Failed Attempt: Algorithm 3

- Shared variables:
  - `boolean flag[2];`  // initially `flag[0] = flag[1] = false`
  - `flag[i] = true` $\Rightarrow P_i$ wants to enter its critical section

- Process $P_i$
  
  ```
  do {flag[i] = true;  // I want to enter
      while (flag[1-i]) ;  // If you also want, then I wait
      critical section
      flag[i] = false;  // I leave
      remainder section
  } while (1);
  ```

- Can satisfy mutual exclusion, but not progress requirement. Why?
Deadlock Problem of Algorithm 3

thread 0
flag[0] = true
while(flag[1] == true):
    wait

deadlock!

thread 1
flag[1] = true
while(flag[0] == true):
    wait
Is the Following Algorithm Correct?

- What if we change the location of the statement: \texttt{flag[i] = true}?

Process \( P_i \):

\[
do { \\
  \text{while (flag[1-\text{i}])} ; \\
  \text{flag[i] = true;} \\
  \text{critical section} \\
  \text{flag[i] = false;} \\
  \text{remainder section} \\
} \text{ while (1);} \\
\]

- Does not satisfy \textbf{mutual exclusion}
Comparison of Algorithms 1, 2, 3

<table>
<thead>
<tr>
<th>Critical Section Algorithms</th>
<th>Test Case 1: $P_0$ serialized enter</th>
<th>Test Case 2: $P_0, P_1$ serialized enter</th>
<th>Test Case 3: $P_0, P_1$ concurrent enter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 1 with a shared lock variable</td>
<td>√</td>
<td>√</td>
<td>× (ME)</td>
</tr>
<tr>
<td>Algorithm 2 with a shared victim variable</td>
<td>× (Progress)</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Algorithm 3 with two shared flag[2] variables</td>
<td>√</td>
<td>√</td>
<td>× (Progress)</td>
</tr>
<tr>
<td>Peterson's Algorithm, with a shared victim variable and two shared flag[2] variables</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Combine the advantages of Algorithms 2 and 3

Operating System Concepts
Peterson's Algorithm

- Combined shared variables of algorithms 2, 3.
- Process $P_i$

```c
    do {
        flag[i] = true;  // I'm interested
        victim = i;      // you go first
        while (flag[1-i] and victim == i) ;
        critical section
        flag[i] = false; // I'm not interested
                        // any more
    } while (1);
```

- Meet all the three requirements; Can solve the critical-section problem for two processes.

Peterson’s Lock: Serialized Acquires

```
flag[0] = true
victim = 0
while(flag[1] == true && victim == 0);

flag[0] = false
```

```
flag[1] = true
victim = 1
while(flag[0] == true && victim == 1);

flag[1] = false
```
Peterson’s Lock: Concurrent Acquires

thread 0

flag[0] = true
victim = 0

while(flag[1] == true && victim == 0);

flag[0] = false

CS₀

thread 1

flag[1] = true
victim = 1

while(flag[0] == true && victim == 1);

flag[1] = false

CS₁

wait
Test the Bounded Waiting Property

Recall: 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.

Test Case: Two processes attempt to enter the critical section simultaneously

- Assume P0 is slow, while P1 is fast
- Can P1 repeatedly grab the exclusive lock, causing P0 to starve?

If yes/no, the solution of critical section cannot/can satisfy bounded waiting property
Proof of Peterson's Algorithm

- The mutual exclusion requirement is assured.
- The progress requirement is assured. The victim variable is only considered when both processes are using, or trying to use, the resource.
- Deadlock is not possible. If both processes are testing the while condition, one of them must be the victim. The other process will proceed.
- Finally, bounded waiting is assured. When a process that has exited the CS reenters, it will mark itself as the victim. If the other process is already waiting, it will be the next to proceed.

https://en.wikipedia.org/wiki/Peterson%27s_algorithm
Quiz: Is the following code correct?

What if we change `victim = i` to `victim = 1-i`?

```c
do {
    flag[i] = true;  // I’m interested
    victim = 1-i;   // I go first
    while (flag[1-i] and victim == i) ;
    critical section
    flag[i] = false; // I’m not interested
    remainder section
} while (1);
```

- Can the code satisfy mutual exclusion?
- Can the code satisfy progress?
- Can the code satisfy bounded waiting?
Quiz: Is the following code correct?

What if we change `victim = i` to `victim = 1-i`?

```c
do {
    flag[i] = true; // I’m interested
    victim = 1-i;   // I go first
    while (flag[1-i] and victim == i) ;
    critical section
    flag[i] = false; // I’m not interested
    remainder section
} while (1);
```

- Can the code satisfy mutual exclusion? NO
- Can the code satisfy progress? YES
- Can the code satisfy bounded waiting? NO
Memory Fence

- Give a C-code demo of Peterson’s algorithm

- A memory barrier, also known as a membar, memory fence or fence instruction, is a type of barrier instruction that causes a central processing unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction.  
  https://en.wikipedia.org/wiki/Memory_barrier

- Operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier.  
Guarantee Memory Access Ordering

- Insert full memory barrier at multiple points

```
    do {
        flag[i] = true;  // I’m interested
        __sync_synchronize(); // full memory barrier
        victim = i;  // You go first
        while (flag[j] and victim == i) ;
        __sync_synchronize(); // full memory barrier
        critical section
        __sync_synchronize(); // full memory barrier
        flag[i] = false; // I’m not interested
        remainder section
    } while (1);
```
Lamport’s Bakery Algorithm

Solve the critical section problem for an arbitrary number of 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, 3, 4, 5...
Bakery Algorithm

- Notation
  - $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - $\max (a_0, \ldots, a_{n-1})$ is a number, $k$, such that $k \geq a_i$ for $i = 0, \ldots, n - 1$

- Shared data
  
  boolean choosing[n];
  int number[n];

  Data structures are initialized to \texttt{false} and \texttt{0} respectively
Bakery Algorithm

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);

Get a ticket first

Which parts are the entry and exit sections?

What is the use of choosing[]?
What if we remove the choosing[]?

do {
    number[i] = max(number[0], number[1], ..., number[n – 1]) + 1;
    for (j = 0; j < n; j++) {
        while ( (number[j] != 0) &&
            ((number[j], j) < (number[i], i)) ) ;
    }
    critical section
    number[i] = 0;
    remainder section
} while (1);

Can the above code satisfy the mutual exclusion and why?

Hint: two processes computed the same number[i].
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware to Build a Lock
- Semaphores
- Classical Problems of Synchronization
- Conditional Variables and 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 (TAS)
  - Swap
  - Atomic fetch-and-add
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 (TAS)

Test and modify the content of a machine word atomically

```java
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
} while(1);
```
Atomic 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 \textbf{false}):  
  \begin{verbatim}
  boolean lock = false;
  \end{verbatim}

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

- Process $P_i$ or Interrupt Handler $TH_i$
  
  do {
    key = true;
    while (key == true)  Swap(lock,key);
    critical section
    lock = false;
    remainder section
  } while(1);

Cannot satisfy bounded waiting. Why?
Another Atomic CPU Instruction Fetch-and-add

**fetch-and-add instruction** performs the operation

```cpp
<< atomic >>
function FetchAndAdd(address location, int inc)
{
    int value := *location
    *location := value + inc
    return value
}
```

**can be used to implement concurrency control structures such as mutex locks and semaphores.**

Bounded Waiting Mutual Exclusion with TestAndSet

- Shared data (initialized to false):
  `boolean lock = false; boolean waiting[n]; //init to false`
- Local variable: `boolean key;`

**Enter Critical Section (Lock)**
```
waiting[i] = true;
key = true;
while (waiting[i] && key)
  key = TestAndSet(lock);
waiting[i] = false;
```

**Leave Critical Section (unlock)**
```
j = (i+1)%n
while ((j!=i) && !waiting[j])
  j = (j+1)%n;
if (j == i)
  lock = false;
else
  waiting[j] = false;
```
Question: What is a spinlock? Why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor system?

- A spinlock is a type of lock that uses busy waiting to acquire the lock. In single-processor systems, this can cause a process that has entered the critical section to be blocked, and another process waiting to enter the critical section to have to wait even longer.
- In multiprocessor systems, when the critical section is short, using a spinlock can be appropriate. With multiple processors, busy waiting by a process does not affect the execution of processes on other processors. Since the critical section is short, the process in the critical section can leave quickly, allowing other busy-waiting processes to enter their critical sections. This avoids the overhead of context switching due to blocking and awakening.

операционные системы

Operating System Concepts
Spin Locks

A spinlock is a lock which causes a thread trying to acquire it to simply wait in a loop ("spin") while repeatedly checking if the lock is available. Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of busy waiting.

```
#include <pthread.h>
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```
An Introduction to Lock-Free Programming

- People often describe **lock-free programming** as programming without kernel-provided mutexes, which are also referred to as locks.

- An Example: DPDK Lockless Ring Buffer
  - Communication between multiple applications running on different CPU cores

![Diagram showing P1 and P2 with a ring buffer and C1 and C2 connections.](http://dpdk.org/doc/guides/prog_guide/ring_lib.html)

- buffer enqueue and dequeue are implemented by atomic Compare-And-Swap (CAS) instruction

---

**Operating System Concepts**

---
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphore as a generic synchronization tool
- Classical Problems of Synchronization
- Conditional Variables and Monitors
- Synchronization Examples
Edsger Wybe Dijkstra

- (Dutch: ['ɛtsxər 'vɪbə 'dɛikstra])
- 11 May 1930 - 6 August 2002 (aged 72)

Known for

- Dijkstra's algorithm (single-source shortest path problem)
- Structured programming, First implementation of ALGOL 60 (“Goto Statements Considered Harmful”)
- Semaphores, Layered approach to operating system design, software-based paged virtual memory in
  THE multiprogramming system
Concept of Semaphore

- It is a synchronization tool that does not require busy waiting, but needs the support from kernel.
  
  `semop()` - Unix, Linux System Call
  http://www.tutorialspoint.com/unix_system_calls/semop.htm

- Semaphore $S$ — an integer variable

- It can only be accessed via two indivisible (atomic) operations: `wait` and `signal`

- They are functionally equivalent to the following busy-waiting operations.

  \[
  \text{wait} (S): \quad \text{while } S \leq 0 \text{ do } \text{no-op;} \\
  \text{ } S--; \\
  \text{signal} (S): \quad S++; \\
  \]
Semaphore Implementation

- Define a semaphore as a record
  
  ```c
  typedef struct {
    int counter;
    struct process *L;
    an in-kernel exclusive lock;
  } semaphore;
  ```

- Assume two simple operations:
  - **block**: block the process that invokes it.
  - **wakeup(P)**: resumes the execution of a blocked process \( P \).
A useful way to think of a semaphore as used in the real-world systems is as a record of how many units of a particular resource are available,
Semaphore operations now defined as follows:

*Uniprocessor solution: disable interrupts. Recall that the only way the dispatcher regains control is through interrupts or through explicit requests.*

**wait(S):**

Disable interrupts;
S.counter--;
if (S.counter < 0) {
    add this process to S.L;
    block;
}
Enable interrupts;

**signal(S):**

Disable interrupts;
S.counter++;
if (S.counter <= 0) {
    remove a process P from S.L;
    wakeup(P);
}
Enable interrupts;

What do we do in a multiprocessor platform to implement $\text{wait}(S)$ and $\text{signal}(S)$?

- Can’t turn off interrupts to get low-level mutual exclusion
- Suppose hardware provides atomic test-and-set instruction

**wait($S$):**

```c
while(TAS(S.lock));
S.counter--;
if (S.counter < 0) {
    add this process to S.L;
    block;
}
S.lock = 0;
```

**signal($S$):**

```c
while(TAS(S.lock));
S.counter++;
if (S.counter <= 0) {
    remove a process $P$ from S.L;
    wakeup(P);
}
S.lock = 0;
```
Semaphore Implementation (3)

```c
Semaphore Implementation (3)

do {
    // spin lock
    while (TestAndSet(lock)) ;
    // spin unlock
    lock = false;
}

// spin
spin

// spin unlock
spin_unlock

// critical section

// remainder section

} while(1);

wait(S):
    S.counter--;
    if (S.counter < 0) {
        add this process to S.L;
        block;
    }

signal(S):
    S.counter++;
    if (S.counter <= 0) {
        remove a process \texttt{P} from S.L;
        wakeup(P);
    }

Use atomic TAS instrument to implement spin lock
```
POSIX Library's Support of Semaphore

- All POSIX semaphore functions and types are prototyped or defined in semaphore.h
  
  ```
  #include <semaphore.h>
  ```

- To define a semaphore object, use
  ```
  sem_t sem_name;   OR   sem_t * sem_pointer;
  ```

- For initialization, use either of the following APIs.
  ```
  int sem_init (sem_t *sem, int pshared, unsigned int initial_value);
  sem_t * sem_open (const char * name, int oflag, unsigned int initial_value);
  ```

- To increment/decrement the value of a semaphore,
  ```
  int sem_wait (sem_t * sem_pointer);
  int sem_post (sem_t * sem_pointer);
  ```
Applications of Binary Semaphore:

1. Act as an Event Notification Tool

- Execute $B$ in $P_j$ only after $A$ executed in $P_i$

- Use semaphore $flag$, which is initialized to 0

Code:

\[
\begin{align*}
  & P_i \quad P_j \\
  & \vdots \quad \vdots \\
  & A \quad \text{wait}(flag) \\
  & \text{signal}(flag) \quad B
\end{align*}
\]
Applications of Binary Semaphore:
2. Solve the Critical Section Problem

- Shared data: `semaphore ex_lock; // initialize = 1`

- Process $P_i$:
  ```do {
  wait(ex_lock);
  critical section
  signal(ex_lock);
  remainder section
  } while (1);```
An example of using semaphore to solve critical section problem

Shared data: semaphore ex_lock; // initialize to 1

Semaphore ex_lock {
    int counter;
    struct process * L;
    an in-kernel exclusive lock;
}

User Mode

进程P1和P2同时竞争ex_lock锁的控制权，可能产生竞争条件。因为都试图修改ex_lock内部的S.counter变量。

Kernel Mode

此时必须依靠内核信号量S的S.lock自旋锁，将wait(S)和signal(S)方法都实现为关键代码段。

wait(S):
    while(TAS(S.lock));
    S.counter--;
    if (S.counter < 0) {
        add this process to S.L;
        block;
    }
    S.lock = 0;

signal(S):
    while(TAS(S.lock));
    S.counter++;
    if (S.counter <= 0) {
        remove a process P from S.L;
        wakeup(P);
    }
    S.lock = 0;
Difference between Binary Semaphore and Mutex

Question: Is there any difference between binary semaphore and mutex, or they are essentially same?

Answer: They're semantically the same, but in practice you will notice weird differences

- Semaphore is implemented by process/thread blocking and wakeup
- Mutex may be internally implemented by some kernels as spin locks, which could be more efficient on multi-processor systems but will slow down a single processor machine
Side Effect of Semaphore: 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.

Example: Let $S$ and $Q$ be two semaphores initialized to 1

\[
P_0 \quad \text{wait}(S); \quad P_0 \quad \text{wait}(Q);
\]
\[
P_0 \quad \text{wait}(Q); \quad \quad \text{wait}(S);
\]
\[
P_0 \quad \text{signal}(S); \quad \quad \text{signal}(Q);
\]
\[
P_0 \quad \text{signal}(Q) \quad \quad \text{signal}(S);
\]

■ **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is waiting.
Chapter 6: Process Synchronization

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

- Bounded-Buffer Problem (or called Producer-Consumer Problem)
- Readers and Writers Problem (or called Shared-Lock Problem)
- Dining-Philosophers Problem
Solution 1 for Producer-Consumer

Shared variables besides the shared buffer

semaphore fillCount, emptyCount;

Initially:

fillCount = 0, emptyCount = n

fillCount: the number of items in the buffer
emptyCount: the number of empty slots in the buffer
Solution 1 for Producer-Consumer

Producer:
do {
    ... 
    produce an item in nextp 
    ... 
    wait(emptyCount);
    add nextp to buffer 
signal(fillCount);
} while (1);

Consumer:
do {
    wait(fillCount);
    remove an item from buffer to nextc 
signal(emptyCount);
        ... 
    consume the item in nextc 
    ... 
} while (1);

Question: Is this solution correct?
Solution 1 for Producer-Consumer

- This solution contains a serious race condition that can result in two or more processes modifying into the same variable `counter` (i.e., the variable recording the number of items in the buffer) at the same time.

- To understand how this is possible, recall how the procedures “add `nextp` to buffer” and “remove an item from buffer” are implemented (by `counter++` and `counter-->`).
Solution 2 for Producer-Consumer

- Shared data

semaphore fillCount, emptyCount, mutex;

Initially:

fillCount = 0, emptyCount = n, mutex = 1

- mutex: guarantee the mutual exclusive access of the shared buffer
Solution 2 for Producer-Consumer

Producer:
do {
    ...
    produce an item in nextp
    ...
    wait(mutex);
    wait(emptyCount);
    add nextp to buffer
    signal(fillCount);
    signal(mutex);
} while (1);

Consumer:
do {
    wait(mutex);
    wait(fillCount);
    remove an item from buffer to nextc
    signal(emptyCount);
    signal(mutex);
    ...
    consume the item in nextc
    ...
} while (1);

Question: Is this solution correct?
Solution 3 for Producer-Consumer

Producer:
do {
    ... 
    produce an item in nextp
    ... 
    wait(emptyCount);
    wait(mutex);
    add nextp to buffer
    signal(mutex);
    signal(fillCount);
} while (1);

Consumer:
do {
    wait(fillCount);
    wait(mutex);
    remove an item from buffer to nextc
    signal(mutex);
    signal(emptyCount);
    ... 
    consume the item in nextc
    ... 
} while (1);
结论1：需要用mutex确保对关键共享资源的互斥访问，比如shared bounded buffer

结论2：信号量wait的顺序很重要
   ◆例子：如果wait(mutex)错误放在了wait(fillCount)或者wait(emptyCount)之前，会导致死锁

问题1：信号量signal的顺序重要吗？
   ◆例子：signal(mutex)和signal(fillCount)可以交换吗？

问题2：能否把produce an item和consume an item放到wait(mutex)和signal(mutex)之间？
When the buffer size is only one, can we remove the mutex variable?

Initially: \[ \text{semaphore } \text{fillCount} = 0, \text{emptyCount} = 1 \]

Producer:
\[
do \{ \\
    \text{produce an item in nextp} \\
    \text{wait(emptyCount);} \\
    \text{add nextp to buffer} \\
    \text{signal(fillCount);} \\
} \text{ while (1);} \\
\]

Consumer:
\[
do \{ \\
    \text{wait(fillCount);} \\
    \text{remove an item from buffer to nextc} \\
    \text{signal(emptyCount);} \\
    \text{consume the item in nextc} \\
} \text{ while (1);} \\
\]

Please give out your reasons.
信号量：生产者消费者习题

考虑三个吸烟者进程和一个经销商进程的系统

每个吸烟者连续不断地做烟卷并抽他做好的烟卷，做一支烟卷需要烟草、纸和火柴三种原料。

这三个吸烟者分别掌握有烟草、纸和火柴。

经销商源源不断地提供上述三种原料，但他只随机的将其中的两种原料放在桌上，具有另一种原料的吸烟者就可以做烟卷并抽烟，且在做完后给经销商发信号，然后经销商再拿出两种原料放在桌上，如此反复。

基于信号量设计一个同步算法描述他们的活动
信号量：生产者消费者习题

可以考虑：设置三个信号量fullA、fullB和fullC，分别代表三种原料组合，初值均为0，即
✿fullA表示烟草和纸的组合，
✿fullB表示纸和火柴的组合，
✿fullC表示烟草和火柴的组合。

桌面上一次只能放一种组合，可以看作是只能放一个产品的共享缓冲区，设置信号量empty初值为1，控制经销商往桌子上放原料
信号量：生产者消费者习题

算法
Semaphore fullA=fullB=fullC=0, empty=1;

process smokerA() {
    do {
        wait(fullA);
        take tobacco and paper from the table;
        signal(empty); // signal an empty table event
        make cigarette;
        smoke cigarette;
    } while (1);
}
process smokerB() {
    do {
        wait(fullB);
        take paper and match from the table;
        signal(empty);
        make cigarette;
        smoke cigarette;
    } while (1);
}

process smokerC() {
    do {
        wait(fullC);
        take tobacco and match from the table;
        signal(empty);
        make cigarette;
        smoke cigarette;
    } while (1);
}
信号量：生产者消费者习题

```c
process provider() {
    integer i;
    do {
        wait(empty); // wait for an empty table event
        i = random() % 3;
        switch(i) {
            case 0: put T&P on table; signal(fullA); break;
            case 1: put P&M on table; signal(fullB); break;
            case 2: put T&M on table; signal(fullC); break;
        }
    } while(1);
}
```
Imagine a number of concurrent operations, including **reads** and **writes**.

- Writes change the state of the data
- Reads do not.

✓ Many reads can proceed concurrently, as long as we can guarantee that no write is on-going.
Readers-Writers Problem (or Shared-Lock Problem)

- Shared data

```c
semaphore mutex, wrt;
int readcount;
```

Initially  

```c
mutex = 1, wrt = 1, readcount = 0
```

- **readcount**: the number of readers browsing the shared content

- **mutex**: guarantee the mutual exclusive access to the readcount variable

- **wrt**: the right of modifying the shared content
Readers-Writers Problem (solution 1)

Writer Process
wait(wrt);
... writing is performed
... signal(wrt);

Question: Is this solution correct?

Reader Process
wait(mutex);
readcount++; signal(mutex);
if (readcount == 1)
    wait(wrt);
... reading is performed ...

wait(mutex);
readcount--; signal(mutex);
if (readcount == 0)
    signal(wrt);
Readers-Writers Problem (solution 2)

Writer Process
wait(wrt);
...
writing is performed
...
signal(wrt);

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);
void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers++;
    if (rw->readers == 1)
        sem_wait(&rw->writelock); // first reader acquires writelock
    sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers--;
    if (rw->readers == 0)
        sem_post(&rw->writelock); // last reader releases writelock
    sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
}

void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(&rw->writelock);
}
More Info about Reader-Writer Locks

The first readers–writers problem
◆ requires that, no reader be kept waiting unless a writer has already obtained permission to use the shared object.

The second readers–writers problem
◆ requires that, once a writer is ready, that writer perform its write as soon as possible.

Discussion:
◆ Which problem is solved by the previous codes?
◆ The first readers-writers problem.
◆ How to solve the second readers-writers problem?
Exercise

用信号量解决无饥饿的读者——写者问题。
The (No-starve) Readers-Writers Problem

```c
void read() {
    do {
        wait(wfmutex);
        signal(wfmutex);
        wait(mutex);
        read_count ++;
        if (read_count == 1)
            wait(writemutex);
        signal(mutex);
        /* reading */
        wait(mutex);
        read_count --;
        if (read_count == 0)
            signal(writemutex);
        signal(mutex);
    } while (1);
}
```

```c
void write() {
    do {
        wait(wfmutex);
        wait(writemutex);
        /* writing */
        signal(writemutex);
        signal(wfmutex);
    } while (1);
}
```

semaphore mutex= 1;
semaphore writemutex=1;
int read_count = 0;
semaphore wfmutex =1;

The (Writer-priority) Readers-Writers Problem

```c
void write() {
    do {
        wait(writecount_mutex);
        write_count ++;
        if (write_count == 1)
            wait(readmutex);
        signal(writecount_mutex);
        wait(writemutex);

        /* writing */
        signal(writemutex);
        wait(writecount_mutex);
        write_count --;
        if (write_count == 0)
            signal(readmutex);
        signal(writecount_mutex);
    } while (1);
}

void read() {
    do {
        wait(readmutex);
        wait(readcount_mutex);
        read_count ++;
        if (read_count == 1)
            wait(writemutex);
        signal(readcount_mutex);
        signal(readmutex);

        /* reading */
        wait(readcount_mutex);
        read_count --;
        if (read_count == 0)
            signal(writemutex);
        signal(readcount_mutex);
    } while (1);
}
```

semaphore readcount_mutex= 1;
semaphore writemutex=1; //0表示不能写
int write_count = read_count = 0;
semaphore writecount_mutex= 1;
semaphore readmutex=1; //0表示不能读
The Dining Philosophers

- Originally formulated in 1965 by Edsger Dijkstra
- Tony Hoare gave the problem its present formulation
Dining-Philosophers Problem

Here is the basic loop of each philosopher:

```c
while (1) {
    think();
    getforks();
    eat();
    putforks();
}
```

- Shared data

  `semaphore chopstick[5];`

Initial values of all semaphores are set to 1
Philosopher $i$:

do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    ...
    eat
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
    think
    ...
} while (1);
Semaphore 学习的四重境界

1. 理解基础概念

2. 熟练掌握经典问题（PC, RW, DP）。

3. 熟悉经典问题的变种，能够将应用题恰当的归约到某个经典问题的变种。

4. 能够将经典问题灵活组合应用，随心所欲，信手拈来。
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Condition Variables and Monitors
- Synchronization Examples
Semaphore and condition variables are very similar and are used mostly for the same purposes.

- Semaphore can be easily understood as an in-kernel counter for the units of a type of resource.
- Condition is an advanced event notification tech.

However, there are minor differences that could make one preferable.

- For example, to implement barrier synchronization, you would not be able to use a semaphore. But a condition variable is ideal.
Condition Variable

- The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true.

Semaphore = counter + mutex + waiting list

Conditional Variable = waiting list

- A problem of semaphore: We cannot read the in-kernel counter hiding inside a semaphore

- A condition variable must be used inside a mutex to avoid a race condition created by one thread preparing to wait and another thread which may signal the condition before the first thread actually waits on it resulting in a deadlock.
<table>
<thead>
<tr>
<th>Semaphore</th>
<th>Condition Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used anywhere</td>
<td>Must be used inside the protection of a mutex</td>
</tr>
<tr>
<td>wait() does not always block its caller</td>
<td>wait() always blocks its caller</td>
</tr>
<tr>
<td>signal() either releases a process, or increases the semaphore counter</td>
<td>signal() either releases a process, or the signal is lost as if it never occurs</td>
</tr>
<tr>
<td>If signal() releases a process, the caller and the released both continue</td>
<td>If signal() releases a process, either the caller or the released continues, but not both</td>
</tr>
</tbody>
</table>
Condition Variable in Pthread Library

Creating/Destroying:
- `pthread_cond_t cond = THREAD_COND_INITIALIZER;`
- `pthread_cond_init`
- `pthread_cond_destroy`

Waiting on condition:
- `pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)` - unlocks the mutex and waits for the condition variable `cond` to be signaled.

Waking thread based on condition:
- `pthread_cond_signal(pthread_cond_t *cond)` - restarts one of the threads that are waiting on the condition variable `cond`.
- `pthread_cond_broadcast(pthread_cond_t *cond)` - wakes up all threads blocked by the specified condition variable.
Barrier Problem

Suppose we wanted to perform a multi-threaded calculation that has two stages, but we don't want to advance to the second stage until the first stage is completed.

We could use a synchronization method called a **barrier**. When a thread reaches a barrier, it will wait at the barrier until all the threads reach the barrier, and then they'll all proceed together.
Barrier Problem

- Pthreads has a `pthread_barrier_wait()` function that implements this. You'll need to declare a `pthread_barrier_t` variable and initialize it with `pthread_barrier_init()`.
  - `pthread_barrier_init()` takes the number of threads that will be participating in the barrier as an argument.

- Now let's implement our own barrier and use it to keep all the threads in sync in a large calculation.
Barrier Implementation by Condition Variable

```c
#define N (16)
double data[256][8192];
pthread_mutex_t m;
pthread_cond_t cv;
int main() {
    int tids[N], i;
    pthread_mutex_init(&m, NULL);
    pthread_cond_init(&cv, NULL);
    for(i = 0; i < N; i++) {
        tids[i] = i;
        pthread_create(&ids[i], NULL, calc, &(tids[i]));
    }
    for(i = 0; i < N; i++) pthread_join(ids[i], NULL);
}
```

double data[256][8192]
void *calc(void *ptr) {

1. Threads do first calculation (use and change values in data)

2. Barrier! Wait for all threads to finish first calculation before continuing

3. Threads do second calculation (use and change values in data)

}
If using condition variable, the state of counter can be access. But when using semaphore, the state of inner count cannot be accessed.

```c
#include <pthread.h>

int remain = N;
void *calc(void *ptr) {
    // The thread does first calculation
    pthread_mutex_lock(&m);
    remain--;
    if (remain == 0) pthread_cond_broadcast(&cv);
    else {
        while(remain != 0) pthread_cond_wait(&cv,&m);
    }
    pthread_mutex_unlock(&m);
    // The thread does second calculation
}```
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 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
- Private Data
  - Monitor Procedure
  - Monitor Procedure
  - Monitor Procedure
- Initialization
Monitors: Event Notification

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`.

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

- The `x.signal` operation wakeup exactly one blocked process. If no process is waiting for the condition, then the `signal` operation has no effect.
Schematic View of a Monitor With Condition Variables
A Subtle Issue of Condition Variable

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 over the monitor and the signaling process waits somewhere.
  - The released process waits somewhere and the signaling process continues to use the monitor.
Java's Monitor Supports

- **Synchronized methods for mutual exclusion**
  ```java
  class classname {
    synchronized return_type methodname() {......}
  }
  ```

- **Coordination support for event notification**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void Object.wait();</td>
<td>Enter a monitor's wait set until notified by another thread</td>
</tr>
<tr>
<td>void Object.wait(long timeout);</td>
<td>Enter a monitor's wait set until notified by another thread or timeout milliseconds elapses</td>
</tr>
<tr>
<td>void Object.notify();</td>
<td>Wake up one thread waiting in the monitor's wait set. (If no threads are waiting, do nothing.)</td>
</tr>
<tr>
<td>void Object.notifyAll();</td>
<td>Wake up all threads waiting in the monitor's wait set. (If no threads are waiting, do nothing.)</td>
</tr>
</tbody>
</table>

Producer-Consumer Example

procedure producer() {
    do {
        item = produceItem();
        PCbuffer.add(item);
    } while (true);
}

procedure consumer() {
    do {
        item = PCbuffer.remove();
        consumeItem(item);
    } while (true);
}

monitor PCbuffer {
    int itemCount; // <= BUFSIZE
    condition full, empty;
    putItemIntoBuffer(item) {...}
    Item removeItemFromBuffer()
    {...}
    procedure void add(item) {
        ...
        // how to implement?
    }
    procedure item remove() {
        ...
        // how to implement?
    }
}
Producer-Consumer Example

procedure void add(item) {
    if (itemCount == BUFSIZE)
        full.wait();
    putItemIntoBuffer(item);
    itemCount = itemCount + 1;
    if (itemCount == 1)
        empty.signal();
    return;
}

procedure item remove() {
    if (itemCount == 0)
        empty.wait();
    item = removeItemFromBuffer();
    itemCount = itemCount - 1;
    if (itemCount == BUFSIZE - 1)
        full.signal();
    return item;
}

Note that if statement has been used in the above code, both when testing if the buffer is full or empty.

With multiple consumers, there is a race condition between the consumer who gets notified that an item has been put into the buffer and another consumer who is waiting on the monitor.
Producer-Consumer Example

procedure void add(item) {
  while (itemCount == BUFSIZE)
    full.wait();
  putItemIntoBuffer(item);
  itemCount = itemCount + 1;
  if (itemCount == 1)
    empty.signal();
  return;
}

procedure item remove() {
  while (itemCount == 0)
    empty.wait();
  item = removeItemFromBuffer();
  itemCount = itemCount - 1;
  if (itemCount == BUFSIZE - 1)
    full.signal();
  return item;
}

- Note that while statement has been used in the above code, both when testing if the buffer is full or empty.

- With multiple consumers, there is a race condition between the consumer who gets notified that an item has been put into the buffer and another consumer who is waiting on the monitor.
Producer-Consumer Example

procedure void add(item) {
    while (itemCount == BUFSIZE)
        full.wait();
    putItemIntoBuffer(item);
    itemCount = itemCount + 1;
    if (itemCount == 1)
        empty.signal();
    return;
}

procedure item remove() {
    while (itemCount == 0)
        empty.wait();
    item = removeItemFromBuffer();
    itemCount = itemCount - 1;
    if (itemCount == BUFSIZE - 1)
        full.signal();
    return item;
}

With multiple producers, there is also a race condition between the producer who gets notified that the buffer is no longer full and another producer is already waiting on the monitor.

If the while was instead an if, too many items might be put into the buffer or a remove might be attempted on an empty buffer.
Dining Philosophers Example

```c
monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i) ;       // pick up chopsticks
    void putdown(int i) ;      // put down chopsticks
    void test(int i) ;         // test if P_i is eligible for eating
    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}
```
### Dining Philosophers Example

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

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

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

The code has NO deadlock!!! Why?

When P₁ and P₄ finish eating at the same time, will P₂ and P₃ compete for their common chopstick after their wakeup?
Another Subtle Issue of Monitor: Queue of Reentering Threads/Proc

A reentering process can be either the released process (from the signaled condition) and the process that signals
For Better Understanding, Let’s Implement Monitor by Semaphores

- Variables
  
  ```c
  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.
Monitor Implementation Using Semaphores

- 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--; 
  ```

- The operation `x.signal` can be implemented as:
  ```
  if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
  }
  ```
Monitor Implementation (Cont.)

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.
Condition Enhanced with a Priority Number

*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.
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Condition Variables and 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.
Adaptive Mutex

Most operating systems (including Solaris, Mac OS X and FreeBSD) use a hybrid approach called "adaptive mutex". The idea is to use a spinlock when trying to access a resource locked by a currently-running thread, but to sleep if the thread is not currently running. (The latter is always the case on single-processor systems.)

https://en.wikipedia.org/wiki/Spinlock#Alternatives
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.