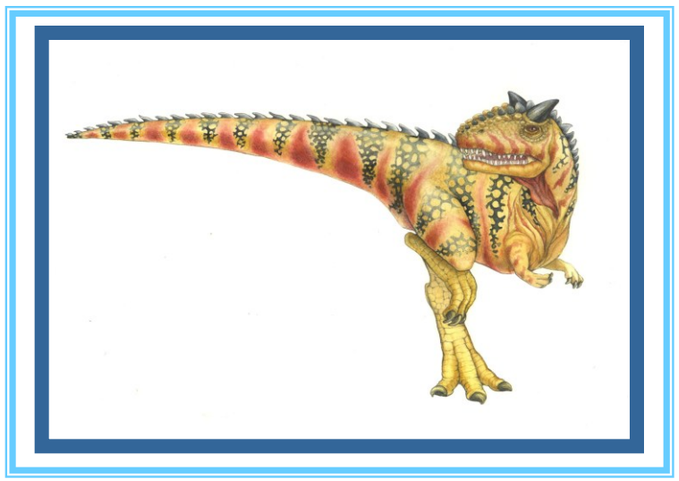
Chapter 6: Process Synchronization



Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
  - Data consistency: the same data stored in different places have the same value
  - Data inconsistency: the same data stored in different places have different values
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

- Illustration of the problem:

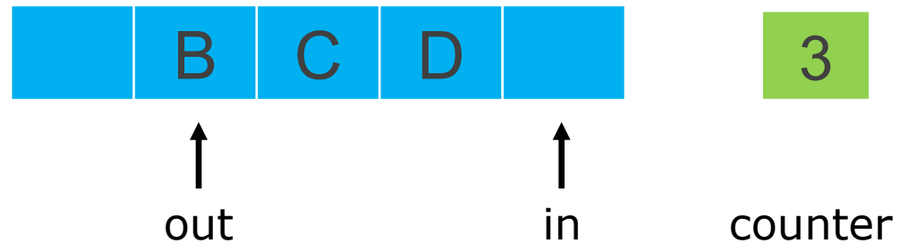
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers.

  We can do so by having an integer **`counter`** that keeps track of the number of full buffers.

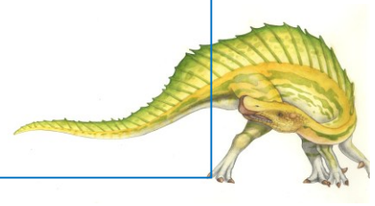
  Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer - Consumer

```
/* Producer */
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE); /* Buffer is full, do nothing */

        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```




```
/* Consumer */
while (true) {
        while (counter == 0); /* Buffer is empty, do nothing */

        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```



increment.c

```
int i=1;

int main()
{
   i++;
}
```

gcc -S increment.c -o increment.s

increment.s

```
i:
        .long   1
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    i(%rip), %eax
        addl    $1, %eax
        movl    %eax, i(%rip)
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

ไม่ใช่คำสั่งเดียว so might be Interrupt

Load value of i into eax register
Add 1 to eax
Store result back in i

Race Condition

Race condition is a situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

Here is example Interrupt

T_0: producer execute `register1 = counter`	{register1 = 5}
T_1: producer execute `register1 = register1 + 1`	{register1 = 6}
T_2: consumer execute `register2 = counter`	{register2 = 5}
T_3: consumer execute `register2 = register2 - 1`	{register2 = 4}
T_4: producer execute `counter = register1`	{counter = 6 }
T_5: consumer execute `counter = register2`	{counter = 4}

Race condition example with 2 processes and shared memory

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024

int main()
{
    int shmid, *count;
    key_t key = 1234;

    // Create shared memory segment
    shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666);

    // Attach to shared memory segment
    count = shmat(shmid, NULL, 0);

    // Initialize shared variable
    *count = 0;

    // Create child processes
    pid_t pid1, pid2;
    pid1 = fork();
    if (pid1 == 0) {
        printf("Child process 1 starts\n");
        // Child process 1 increases the shared variable 100,000 times
        int i;
        for (i = 0; i < 100000; i++) {
                (*count)++;
        }
        exit(0);
    }
    pid2 = fork();
    if (pid2 == 0) {
        printf("Child process 2 starts\n");
        // Child process 2 decreases the shared variable 100,000 times
        int i;
        for (i = 0; i < 100000; i++) {
                (*count)--;
        }
        exit(0);
    }

    // Wait for child processes to finish
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);

    // Print final value of shared variable
    printf("Final count: %d\n", *count);

    // Detach from shared memory segment
    shmdt(count);

    // Remove shared memory segment
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

```
veera@Yoga:~/OS$ ./race_shmem
Child process 1 starts
Child process 2 starts
Final count: 0
veera@Yoga:~/OS$ ./race_shmem
Child process 1 starts
Child process 2 starts
Final count: -28802
veera@Yoga:~/OS$ ./race_shmem
Child process 2 starts
Child process 1 starts
Final count: 68108
```

```
veera@Yoga:~/OS$ ./race_shmem
Child process 1 starts
Child process 2 starts
Final count: 0
veera@Yoga:~/OS$ ./race_shmem
Child process 1 starts
Child process 2 starts
Final count: 48672
veera@Yoga:~/OS$ ./race_shmem
Child process 1 starts
Child process 2 starts
Final count: 0
```

Race condition example with 2 threads

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int count = 0;

void *thread_function1(void *arg)
{
    int i;
    printf("Thread 1 starts\n");
    // Thread 1 increases the shared variable 100,000 times
    for (i = 0; i < 100000; i++) {
        count++;
    }
    return NULL;
}

void *thread_function2(void *arg)
{
    int i;
    printf("Thread 2 starts\n");
    // Thread 2 decreases the shared variable 100,000 times
    for (i = 0; i < 100000; i++) {
        count--;
    }
    return NULL;
}
```

```c
int main()
{
    pthread_t thread1, thread2;

    // Create threads
    pthread_create(&thread1, NULL, thread_function1, NULL);
    pthread_create(&thread2, NULL, thread_function2, NULL);

    // Wait for threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);


    // Print final value of count
    printf("Final count: %d\n", count);

    return 0;
}
```

```
veera@Yoga:~/OS$ ./race_threads
Thread 1 starts
Thread 2 starts
Final count: 0
veera@Yoga:~/OS$ ./race_threads
Thread 1 starts
Thread 2 starts
Final count: 1320
veera@Yoga:~/OS$ ./race_threads
Thread 1 starts
Thread 2 starts
Final count: -28133
veera@Yoga:~/OS$ ./race_threads
Thread 1 starts
Thread 2 starts
Final count: 27898
veera@Yoga:~/OS$ ./race_threads
Thread 1 starts
Thread 2 starts
Final count: -17666
```

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

part of code that Race Condition occur

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

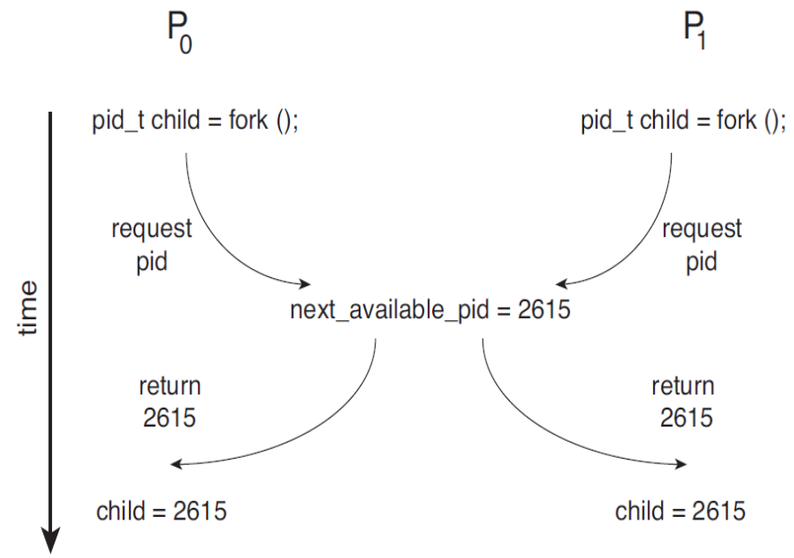
Solution to Critical-Section Problem

A solution to critical-section problem must satisfy these requirements:

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

Critical-Section Handling in OS

- Multiple processes may concurrently enter kernel mode and access kernel data structures for maintaining opened files, memory allocation, processes, interrupt handling, etc.
- Kernel data structures are prone to possible race conditions.
- Solution in a single-core environment: disable interrupts while a shared variable is being modified
  - Infeasible in multiprocessor systems



Race condition when assigning a PID for a new process

- Two approaches to handle critical sections in OS:
  - **Non-preemptive kernel** – kernel-mode process runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode
  - **Preemptive kernel** – kernel-mode process can be preempted
    - Multiple processes can run in kernel-mode on different processors.

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptible
  - test-and-set
  - compare-and-swap
  - fetch-and-add, fetch-and-sub

The definition of the atomic `test_and_set()` instruction.

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Mutual-exclusion implementation with `test_and_set()`.

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

If lock is TRUE, set LOCK to TRUE (unchanged) and return TRUE. So, repeat the loop.

If lock is FALSE, set lock to TRUE and return FALSE. So, exit the loop and enter critical section.

increment_atomic.c

```
#include <stdatomic.h>

atomic_int i = 1;

int main() {
    atomic_fetch_add(&i, 1);
    return 0;
}
```

gcc -S increment_atomic.c
-o increment_atomic.s

increment_atomic.s

```
i:
        .long   1
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        lock addl       $1, i(%rip)  # Atomic fetch-and-add
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Solve a race condition with atomic operation

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024

int main()
{
    int shmid, *count;
    key_t key = 1234;

    // Create shared memory segment
    shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666);

    // Attach to shared memory segment
    count = shmat(shmid, NULL, 0);

    // Initialize shared variable
    *count = 0;

    // Create child processes
    pid_t pid1, pid2;
    pid1 = fork();
    if (pid1 == 0) {
        printf("Child process 1 starts\n");
        // Child process 1 increments the shared variable 100,000 times
        int i;
        for (i = 0; i < 100000; i++) {
            // Enter critical section
            __sync_fetch_and_add(count, 1);
            // Exit critical section
        }
        exit(0);
    }
```

```c
    pid2 = fork();
    if (pid2 == 0) {
        printf("Child process 2 starts\n");
        // Child process 2 decreases the shared variable 100,000 times
        int i;
        for (i = 0; i < 100000; i++) {
            // Enter critical section
            __sync_fetch_and_sub(count, 1);
            // Exit critical section
        }
        exit(0);
    }

    // Wait for child processes to finish
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);

    // Print final value of shared variable
    printf("Final count: %d\n", *count);

    // Detach from shared memory segment
    shmdt(count);

    // Remove shared memory segment
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

```
veera@Yoga:~/OS$ ./race_shmem_atomic
Child process 1 starts
Child process 2 starts
Final count: 0
veera@Yoga:~/OS$ ./race_shmem_atomic
Child process 1 starts
Child process 2 starts
Final count: 0
veera@Yoga:~/OS$ ./race_shmem_atomic
Child process 1 starts
Child process 2 starts
Final count: 0
```

Mutex Locks

- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
 }

 release() {
    available = true;
 }

do {
acquire lock
    critical section
release lock
  remainder section
} while (true);
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore ***S*** – integer variable
- Two standard operations modify ***S***: **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

```
signal (S) {
    S++;
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Similar to a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore

- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

```
P1:
    S1;
    signal(synch);

P2:
    wait(synch);
    S2;
```

statement

binary semaphore, begin with 0

so P2 will wait until S1 finish → signal

synch → 1 → can run P2

Semaphore Implementation

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation

```
typedef struct{
    int value;
    struct process *list;
} semaphore;

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Semaphore Implementation

- Must guarantee that no two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time

- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied

- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Solving a race condition between processes with a semaphore

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>

#define SHM_SIZE 1024

int main()
{
    int shmid, *count;
    key_t key = 1234;

    // Create shared memory segment
    shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666);

    // Attach to shared memory segment
    count = shmat(shmid, NULL, 0);

    // Initialize shared variable
    *count = 0;

    // Create semaphore
    sem_t *sem;
    sem = sem_open("/my_semaphore", O_CREAT, 0644, 1);

    // Create child processes
    pid_t pid1, pid2;
    pid1 = fork();
    if (pid1 == 0) {
        printf("Child process 1 starts\n");
        // Child process 1 increments the shared variable 100,000 times
        int i;
        for (i = 0; i < 100000; i++) {
            sem_wait(sem);
            (*count)++;
            sem_post(sem);
        }
        exit(0);
    }
    pid2 = fork();
    if (pid2 == 0) {
        printf("Child process 2 starts\n");
        // Child process 2 decreases the shared variable 100,000 times
        int i;
        for (i = 0; i < 100000; i++) {
            sem_wait(sem);
            (*count)--;
            sem_post(sem);
        }
        exit(0);
    }

    // Wait for child processes to finish
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);

    // Print final value of shared variable
    printf("Final count: %d\n", *count);

    // Detach from shared memory segment
    shmdt(count);

    // Remove shared memory segment
    shmctl(shmid, IPC_RMID, NULL);

    // Close and unlink semaphore
    sem_close(sem);
    sem_unlink("/my_semaphore");

    return 0;
}
```

```
veera@Yoga:~/OS$ ./race_shmem_atomic
Child process 1 starts
Child process 2 starts
Final count: 0
veera@Yoga:~/OS$ ./race_shmem_atomic
Child process 1 starts
Child process 2 starts
Final count: 0
veera@Yoga:~/OS$ ./race_shmem_atomic
Child process 1 starts
Child process 2 starts
Final count: 0
```

Solving a race condition between threads with a mutex

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int count = 0;
pthread_mutex_t mutex;

void *thread_function1(void *arg)
{
    int i;
    printf("Thread 1 starts\n");
    // Thread 1 increases the shared variable 100,000 times
    for (i = 0; i < 100000; i++) {
        pthread_mutex_lock(&mutex); // Acquire the lock
        count++;
        pthread_mutex_unlock(&mutex); // Release the lock
    }
    return NULL;
}

void *thread_function2(void *arg)
{
    int i;
    printf("Thread 2 starts\n");
    // Thread 2 decreases the shared variable 100,000 times
    for (i = 0; i < 100000; i++) {
        pthread_mutex_lock(&mutex); // Acquire the lock
        count--;
        pthread_mutex_unlock(&mutex); // Release the lock
    }
    return NULL;
}
```

```c
int main()
{
    pthread_t thread1, thread2;

    // Initialize the mutex
    pthread_mutex_init(&mutex, NULL);

    // Create threads
    pthread_create(&thread1, NULL, thread_function1, NULL);
    pthread_create(&thread2, NULL, thread_function2, NULL);

    // Wait for threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Destroy the mutex
    pthread_mutex_destroy(&mutex);

    // Print final value of count
    printf("Final count: %d\n", count);

    return 0;
}
```

```
veera@Yoga:~/OS$ gcc -o race_threads_mutex race_threads_mutex.c -pthread
veera@Yoga:~/OS$ ./race_threads_mutex
Thread 1 starts
Thread 2 starts
Final count: 0
veera@Yoga:~/OS$ ./race_threads_mutex
Thread 1 starts
Thread 2 starts
Final count: 0
```

Deadlock and Starvation

Problem in Semaphore

- **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);
```

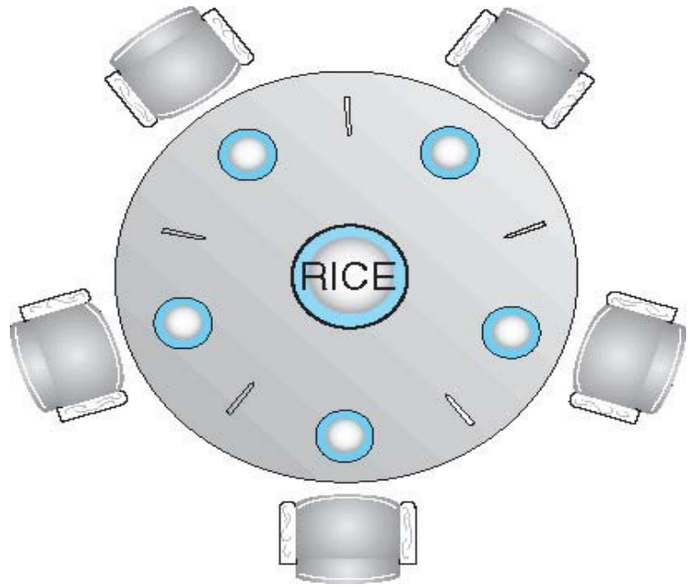
Deadlock here

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**
  - All processes that are accessing resources needed by a higher-priority process tempolarily inherit the higher priority until they are finished with the resources.

Classical Problems of Synchronization

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

Dining-Philosophers Problem



- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

ต้องใช้ lock (semaphore) มาช่วย to access

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do  {
        wait ( chopstick[i] );
         wait ( chopStick[ (i + 1) % 5] );

                // eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

            // think

} while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

Linux Synchronization

- Linux provides:
  - Atomic integers
  - Mutex locks
  - Semaphores
  - Spinlocks

- On a single-cpu system, spinlocks are replaced by enabling and disabling kernel preemption.