LINUX PROGRAMMING

Thread and Concurrency

www.cce.hcmut.edu.vn

06/10/2022

Le Thanh Van

Outline

- Race condition
- The Critical-Section Problem
- Peterson’s Solution
- Mutex Locks
- Semaphores
- Monitor
- Condition variable

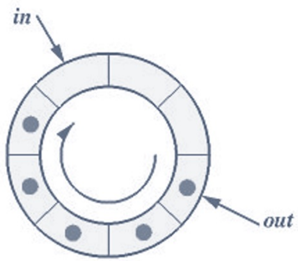
Background

- Processes can execute *concurrently* (or *in parallel*)
  - May be interrupted at any time, partially completing execution
- Concurrent access to *shared data* may result in *data inconsistency*
- Maintaining data consistency requires mechanisms to ensure the *orderly execution of cooperating processes*
- Illustration of the problem:
  - Consider a solution to the *consumer-producer problem* that fills *all* the buffers. Use 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 adds a new item to the buffer and is *decremented* by the consumer after it consumes an item from the buffer

```
#define BUFFER_SIZE 8
            /* 8 buffers
*/

typedef struct {
            . . .
} item;
item buffer[BUFFER_SIZE];

int in = 0;
int out = 0;
int counter = 0;
```

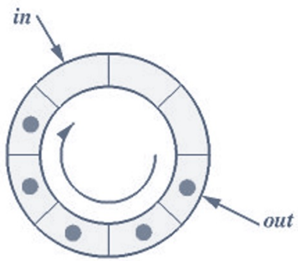
Producer

```
while (true) {
        /* produce an item in next produced */
          while (counter == BUFFER_SIZE)
                    ;         /* do nothing */
          buffer[in] = next_produced;
          in = (in + 1) % BUFFER_SIZE; /* pointer in to buffer */
          counter++;
}
```



Consumer

```
while (true) {
          while (counter == 0)
                    ;         /* do nothing */
          next_consumed = buffer[out];
          out = (out + 1) % BUFFER_SIZE;/* pointer out from buffer */
        counter--;
          /* consume the item in next_consumed */
}
```



Race Condition

- `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 "`counter = 5`" initially:

S0: producer execute `register1 = counter` {register1 = 5}

S1: producer execute `register1 = register1 + 1` {register1 = 6}

S2: consumer execute `register2 = counter` {register2 = 5}

S3: consumer execute `register2 = register2 – 1` {register2 = 4}

S4: producer execute `counter = register1` {counter = 6 }

S5: consumer execute `counter = register2` {counter = 4}

=> Data inconsistency

Critical-Section Problem

- Consider system of ***n*** processes $\{P_0, P_1, \ldots P_{n-1}\}$
- Each process has *critical section* (i.e., 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* needs to design a 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*

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

Critical Section (CS)

- General structure of the process P_i

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

Exercise (1)

- #define MAX_RESOURCES 5
- int available_resources = MAX_RESOURCES;

```
/* decrease available_resources by
count resources   return 0 if sufficient
resources available,
 otherwise return -1  */
int decrease_count(int count) {
if (available_resources < count)
return -1;
else {
available_resources -=count;
return 0;
}
}
```

```
/* increase available_resources by count */
int increase_count(int count) {
available_resources += count;
return 0;
}
```

1. Identify the data involved in the race condition.
2. Identify the location (or locations) in the code where the race condition occurs.

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 process 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

Proposal solution 1 (1/2)

- Shared variable
  - **int turn**; /* initialize **turn = 0** */
  - If **turn =** i then P_i is permitted to enter CS
- Process P_i

```
do {
        while (turn != i);
        critical section
        turn = j;
        remainder section
} while (1);
```

Proposal solution 1 (2/2)

```
Process P0
do {
      while (turn != 0);
      critical section
      turn := 1;
      remainder section
} while (1);
```

```
Process P1
do {
      while (turn != 1);
      critical section
      turn := 0;
      remainder section
} while (1);
```

- Achieve mutual exclusion (1),
- Violate condition of progress (2).

Proposal solution 2 (1/2)

- Shared variable
  - boolean flag[2]; /* initialize flag[0] = flag[1] = **false** */
  - **flag[i] = true** notice that P_i want to enter CS
- Process P_i

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

- Achieve mutual exclusion (1),
- Violate condition of progress (2).

Proposal solution 2 (2/2)

- Process P_0

```
do {
        flag[ 0 ] = true;
        while (flag[ 1 ]);
        critical section
        flag[ 0 ] = false;
        remainder section
} while (1);
```

- Process P_1

```
do {
        flag[ 1 ] = true;
        while (flag[ 0 ]);
        critical section
        flag[ 1 ] = false;
        remainder section
} while (1);
```

- Achieve mutual exclusion (1),
- Violate condition of progress (2).

Peterson's Solution

- Not guaranteed to work on modern architectures!
  - (But good algorithmic description of solving the problem)
- *Two-processes* solution
- Assume that the **load** and **store** machine-language instructions are *atomic*; that is, it cannot be interrupted
- The two processes share *two variables*:
  - **int turn;**
  - **boolean flag[i]**
    - The variable **turn** indicates whose turn it is to enter the critical section
    - The **flag[]** array is used to indicate if a process is ready to enter the critical section
      - **flag[i] = *true*** implies that process P_i is ready!

Algorithm for Process P_i

```
while (true){
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;       /* do nothing */
    /* critical section */
    flag[i] = false;
    /* remainder section */

}
```

```
P0: while (flag[1] && turn == 1);

    /* do nothing */
```

```
P1: while (flag[0] && turn == 0);

    /* do nothing */
```

Peterson's Solution (Cont.)

Provable that the three CS requirement are met:

Mutual exclusion is preserved

- P_i enters CS only if: either **`flag[j] = false`** or **`turn = i`**

Progress requirement is satisfied

Bounded-waiting requirement is met

Remarks on Peterson’s Solution

- Although useful for demonstrating an algorithm, Peterson’s Solution is *not guaranteed to work on modern architectures*
- Understanding why it will not work is also useful for better understanding *race conditions*
- To improve performance, processors and/or compilers may *reorder operations* that have no dependencies
  - For *single-threaded,* this is ok as the result will always be the same.
  - For *multithreaded,* the reordering may produce inconsistent or unexpected results!

Example of Peterson’s Solution

- Two threads share the data:

```
boolean flag = false;
int x = 0;
```

- *Thread 1* performs

```
while (!flag)
 ;
print x
```

- *Thread 2* performs

```
x = 100;
flag = true
```

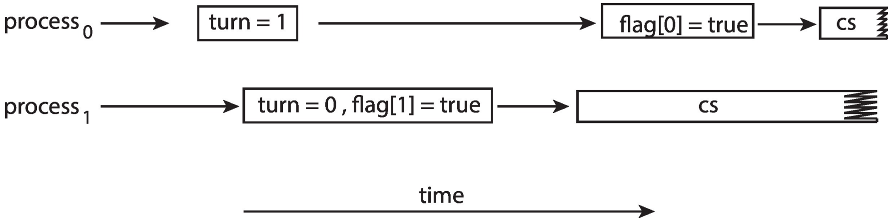
- What is the expected output?

Example of Peterson's Solution

- 100 is the expected output.
- However, the operations for *Thread 2* may be reordered:

```
flag = true;
x = 100;
```

- If this occurs, the output may be 0!
- The effects of *instruction reordering* in Peterson's Solution



- This allows both processes to be in their critical section at the same time!

Atomic Variables

- *atomic variable* provides *atomic* (uninterruptible) updates on basic data types such as Integers and Booleans.
- For example, the **`increment()`** operation on the atomic variable **`sequence`** ensures **`sequence`** is incremented without interruption:

```
increment(&sequence);
```

Atomic Variable example

```c
atomic_int acnt;
int cnt;

void* f(void* thr_data)
{
    (void)thr_data;
    for(int n = 0; n < 1000; ++n) {
        ++cnt;
        ++acnt;
    }
    return 0;
}

int main(void)
{
    pthread_t thr[10];
    int ret;
    for(int n = 0; n < 10; ++n)
        pthread_create(&thr[n], NULL,f, NULL);
    for(int n = 0; n < 10; ++n)
        ret=pthread_join(thr[n], NULL);

    printf("The atomic counter is %u\n", acnt);
    printf("The non-atomic counter is %u\n", cnt);

}
```

What is the expected value of acnt?
What is the expected value of cnt?

Using *test_and_set* Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter (i.e., **`*target`**)
3. Set the new value of passed parameter to **`true`**
   (i.e., **`*target=true`**)

Solution using *test_and_set()*

- Shared Boolean variable **lock**, initialized to **false**
- Solution:

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

       /* critical section */

  lock = false;
       /* remainder section */
} while (true);
```

Mutex Locks and Spinlock

- *Previous solutions are complicated* and generally inaccessible to application programmers
- OS designers build *software tools* to solve critical section problem
- Simplest is **mutex lock**
- 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*
- But this solution requires *busy waiting*
  - This lock therefore called a *spinlock*

Solution to Critical-section Problem using Locks

```
while (true) {
          acquire lock;

          critical section;

          release lock;

          remainder section;
}
```

Mutex Lock Definitions

- **acquire() {**
```
    while (!available)
      ; /* busy wait */
    available = false;;
  }
```
- **release() {**
```
    available = true;
  }
```
- These two functions must be implemented *atomically*

Semaphore

- *Synchronization tool* that provides more sophisticated ways (than mutex locks) for process to synchronize their activities.
- Semaphore **S** is an integer variable
- Can only be accessed via *two* indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - (Originally called **`P()`** and **`V()`**)

Semaphore Implementation

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

Semaphore Implementation with no Busy waiting

- 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

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

Implementation with no Busy waiting (Cont.)

```
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);
    }
}
```

2. Exercise 2

```
semaphore S1, S2;
S1.value = 1;
S2.value = 0;
```

Process P1:

```
while (1) {
    wait(S1);
    Critical section
    signal(S2);
}
```

Process P2:

```
while (1) {
    wait(S2);
    Critical section
    signal(S2);
}
```

Determine the order of execution in critical section of P1 and P2?

Semaphore Usage

- *Counting semaphore* – integer value can range over an unrestricted domain
- *Binary semaphore* – integer value can range only between 0 and 1
  - Same as a mutex lock
  - Can solve various synchronization problems
- Can implement a counting semaphore S as a binary semaphore
  - Consider P_1 and P_2 that require S_1 to happen before S_2
    - Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Problems with Semaphores

- Incorrect use of **semaphore** operations:
  - `signal(mutex)` … `wait(mutex)`
  - `signal(mutex)` … `signal(mutex)`
  - `wait(mutex)` … `wait(mutex)`
  - Omitting of `wait(mutex)` and/or `signal(mutex)`

- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

An example of Semaphore problem

```
int thread_flag;
pthread_mutex_t thread_flag_mutex;
void initialize_flag ()
{
  pthread_mutex_init (&thread_flag_mutex, NULL);
  thread_flag = 0;
}
```

```
void set_thread_flag (int flag_value)
{
    /* Protect the flag with a mutex lock. */
    pthread_mutex_lock (&thread_flag_mutex);
        thread_flag = flag_value;
    pthread_mutex_unlock (&thread_flag_mutex);

}
```

```
void* thread_function (void* thread_arg)
{
    while (1) {
        int flag_is_set;
        /* Protect the flag with a mutex lock. */
        pthread_mutex_lock (&thread_flag_mutex);
            flag_is_set = thread_flag;
        pthread_mutex_unlock (&thread_flag_mutex);
        if (flag_is_set)
            do_work ();
        /* Else don't do anything. Just loop again. */
}
    return NULL;
}
```

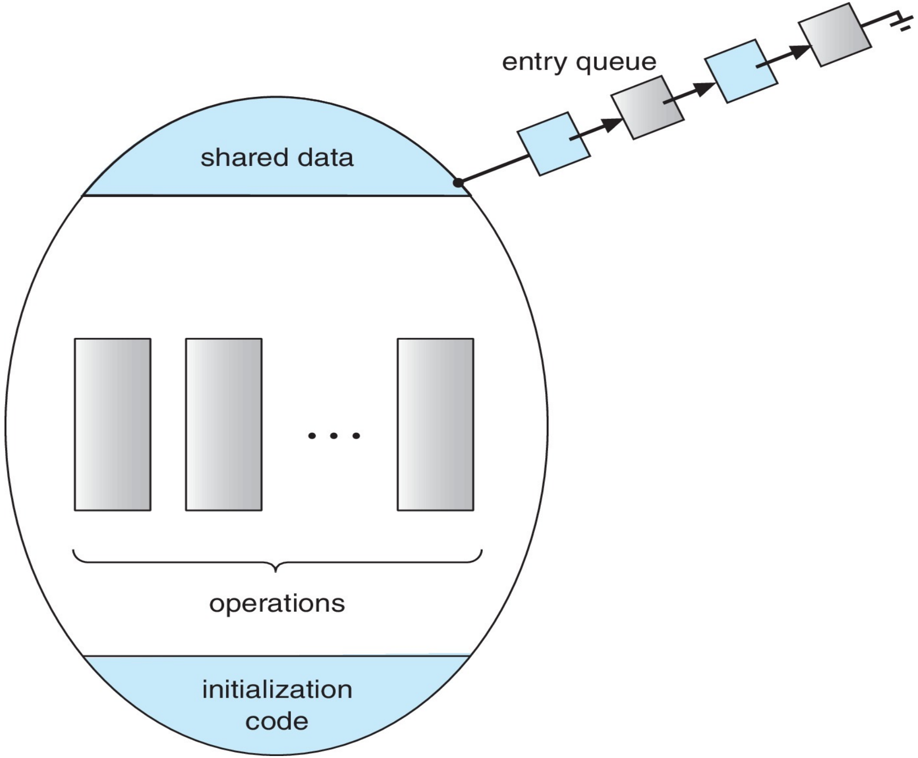
May cause starvation?

Monitors

- A *high-level abstraction* that provides a convenient and effective mechanism for process synchronization
- Abstract *data type*, *internal variables* only accessible by code within the procedure
- *Only one process* may be active within the monitor at a time
- Pseudocode syntax of a **monitor**:

```
monitor monitor-name
{
    // shared variable declarations
    function P1 (…) { …. }
    function P2 (…) { …. }
    function Pn (…) {……}
   initialization code (…) { … }
}
```

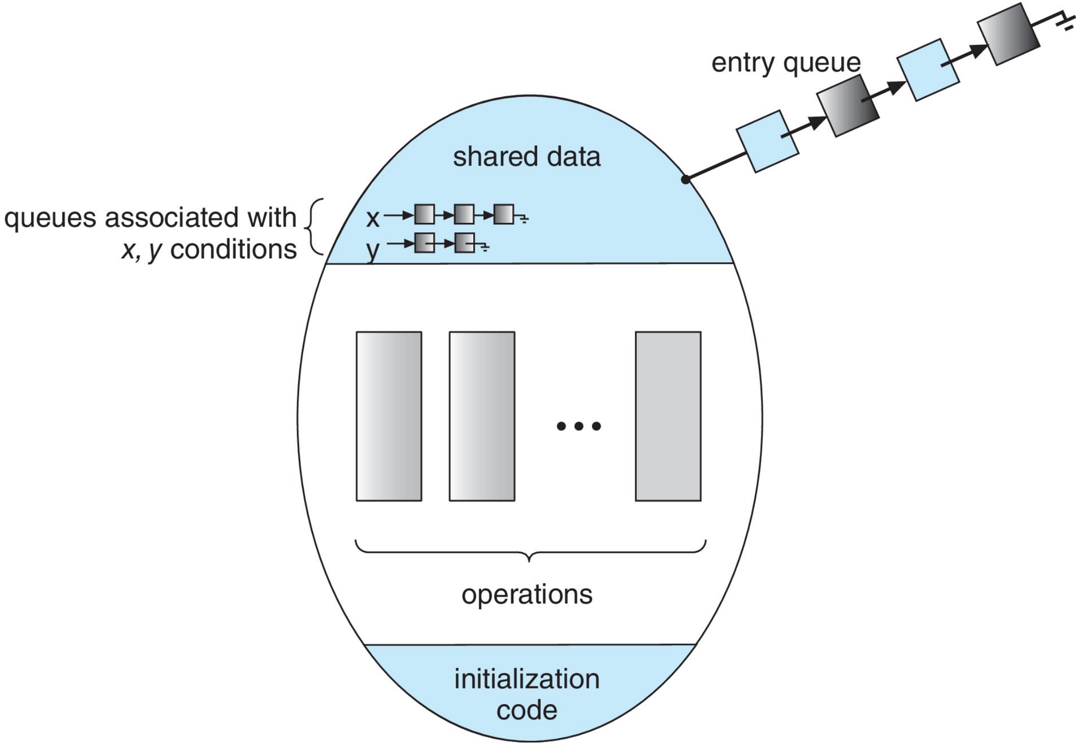
Schematic View of a Monitor



Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process **P** invokes **`x.signal()`**, and process **Q** is suspended in **`x.wait()`**, what should happen next?
  - Both **Q** and **P** can't execute in parallel. If **Q** is resumed, then **P** must wait
- Options include
  - ***Signal and wait*** – **P** waits until **Q** either leaves the monitor or it waits for another condition
  - ***Signal and continue*** – **Q** waits until **P** either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in **Concurrent Pascal** compromise
    - **P** executing *signal* immediately leaves the monitor, **Q** is resumed
  - Implemented in other languages: **Mesa**, **C#**, **Java**

Producer-Consumer with Monitors (Hoare)

```
Monitor bounded_buffer {
  buffer resources[N];
  condition not_full, not_empty;
produce(resource x) {
    if (array "resources" is full, determined maybe by a count)
        wait(not_full);
    insert "x" in array "resources"
    signal(not_empty);
 }
        consume(resource *x) {
    if (array "resources" is empty, determined maybe by a count)
        wait(not_empty);
    *x = get resource from array "resources"     signal(not_full);
 }
```

Producer-Consumer with Monitors (Mesa)

```
Monitor bounded_buffer {
  buffer resources[N];
  condition not_full, not_empty;
produce(resource x) {
    while (array "resources" is full, determined maybe by a count)
          wait(not_full);
    insert "x" in array "resources"
    signal(not_empty);
  }
         consume(resource *x) {
    while (array "resources" is empty, determined maybe by a count)
          wait(not_empty);
    *x = get resource from array "resources"     signal(not_full);
  }
```

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 **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

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

Semaphore **mutex** = 1
Semaphore **full** = 0
Semaphore **empty** = n

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

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

Semaphore **mutex** = 1
Semaphore **full** = 0
Semaphore **empty** = n

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

```
while (true) {
        wait(rw_mutex);
          ...
        /* writing is performed */
          ...
     signal(rw_mutex);
}
```

Semaphore **rw_mutex** = 1

Semaphore **mutex** = 1

Integer **read_count** = 0

Readers-Writers Problem (Cont.)

Semaphore **rw_mutex** = 1

Semaphore **mutex** = 1

Integer **read_count** = 0

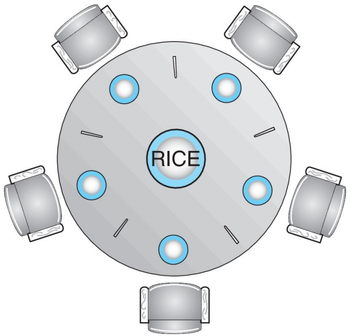
```
while (true) {
        wait(rw_mutex);
                ...
          /* writing is performed
*/
                ...

        signal(rw_mutex);
}
```

- The structure of a reader process

```
while (true){
        wait(mutex);
        read_count++;
        if (read_count == 1)
        wait(rw_mutex);
        signal(mutex);
    ...   /* reading is performed */
    ...
        wait(mutex);
        read count--;
        if (read_count == 0)
                signal(rw_mutex);
        signal(mutex);
}
```

Dining-Philosophers Problem

- Philosophers spend their lives alternating 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



Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher *i*:

```
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?

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

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and releasing the lock

```
/* 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:

```
#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:

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

/* critical section */

/* release the semaphore */
sem_post(sem);
```

POSIX Unnamed Semaphores

- Creating an initializing the semaphore:

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

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

- Acquiring and releasing the semaphore:

```
/* 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:

```
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:

```
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:

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

Condition variable in POXIS thread library

- CV: condition variable
- pthread_cond_t cond; //declare an instance of CV.
- pthread_mutex_t mutex; //declare an instance of mutex
- pthread_cond_init(&cond, NULL); //initialize a CV
- pthread_cond_signal(&cond); //signal a CV: unblock a thread that is blocked on this CV. If no thread is blocked on the CV, the signal is ignore.
- pthread_cond_broadcast(&cond); //unblock all threads that are blocked on this CV.
- pthread_cond_wait(&cond,&mutex): Before calling this function, the mutex must be locked by the calling thread. When calling this function, it unlocks the mutex and blocks on the CV.
  - When the CV is signaled and the calling thread unblocks, pthread_cond_wait reacquires a lock on mutex.

Condition variable … (cont)

```
void initialize_flag (){
  pthread_mutex_init (&thread_flag_mutex, NULL);
  pthread_cond_init (&thread_flag_cv, NULL);
  thread_flag = 0;
}
```

```
void set_thread_flag (int flag_value) {
    pthread_mutex_lock (&thread_flag_mutex);
        thread_flag = flag_value;
        pthread_cond_signal (&thread_flag_cv);
    pthread_mutex_unlock (&thread_flag_mutex);
}
```

```
void* thread_function (void* thread_arg){
    while (1) {
       pthread_mutex_lock (&thread_flag_mutex);
       while (!thread_flag)
            pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
      pthread_mutex_unlock (&thread_flag_mutex);
      do_work ();
}
    return NULL;

}
```

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a *mutex lock* or *semaphore*
- Waiting indefinitely violates the *progress* and *bounded-waiting* criteria discussed at the beginning of this chapter
- **Liveness** refers to a *set of properties* that a system must satisfy to ensure processes make progress
- Indefinite waiting is an example of a liveness failure

Liveness (Cont.)

- *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 P_1

```
wait(S);          wait(Q);
wait(Q);          wait(S);
 ...               ...
signal(S);        signal(Q);
signal(Q);        signal(S);
```

- Consider if P_0 executes **`wait(S)`** and P_1 **`wait(Q)`**. When P_0 executes **`wait(Q)`**, it must wait until P_1 executes **`signal(Q)`**
- However, P_1 is waiting until P_0 execute **`signal(S)`**
- Since these **`signal()`** operations will never be executed, P_0 and P_1 are *deadlocked*

Exercise (3)

```
monitor resources
{
        int available_resources;
        condition resources_avail;

        int decrease_count(int count)
        {
          IF/WHILE (available_resources < count)
                            resources_avail.wait();
                available_resources = available_resources - count;
        }
        int increase_count(int count)
        {
                available_resources = available_resources + count;
                resources_avail.signal();
        }
}
```

What's the problem with the given code?

THANK YOU !

Center Of Computer Engineering

www.cce.hcmut.edu.vn