Operating System

Nguyen Tri Thanh
ntthanh@vnu.edu.vn
Review

- A system uses FCFS process (arrived_time, duration)
  - $P_1(0,20)$, $P_2(30,10)$, $P_3(20,40)$, $P_4(50,15)$
- Which of the following is the correct running order of the above processes?
  A. $P_1$, $P_2$, $P_3$, $P_4$
  B. $P_1$, $P_3$, $P_2$, $P_4$
  C. $P_1$, $P_4$, $P_2$, $P_3$
  D. $P_5$, $P_2$, $P_3$, $P_1$
A system uses SJF process (arrived\_time, duration)

- $P_1(0,20)$, $P_2(30,10)$, $P_3(20,40)$, $P_4(50,15)$

Which of the following is the correct running order of the above processes?

A. $P_1, P_2, P_3, P_4$
B. $P_1, P_4, P_2, P_3$
C. $P_1, P_3, P_2, P_4$
D. $P_4, P_2, P_3, P_1$
A system uses SRTF process (arrived_time, duration)

- $P_1(0,20)$, $P_2(30,10)$, $P_3(20,40)$, $P_4(40,15)$

Which of the following is the correct running order of the above processes?

A. $P_1, P_3, P_2, P_4, P_3$
B. $P_1, P_2, P_3, P_4, P_4$
C. $P_1, P_4, P_2, P_3, P_2$
D. $P_1, P_2, P_3, P_1, P_4$
Review

• A system uses RR process (arrived_time, duration)
  • $P_1(0,22)$, $P_2(30,10)$, $P_3(20,40)$, $P_4(40,25)$
  • Time quantum = 15

• Which of the following is the correct running order of the above processes?
  A. $P_1$, $P_2$, $P_3$, $P_1$, $P_2$, $P_3$, $P_4$, $P_3$
  B. $P_1$, $P_3$, $P_1$, $P_3$, $P_2$, $P_3$, $P_4$, $P_3$
  C. $P_1$, $P_1$, $P_2$, $P_3$, $P_2$, $P_3$, $P_4$, $P_3$
  D. $P_1$, $P_1$, $P_3$, $P_2$, $P_3$, $P_4$, $P_3$, $P_4$
A system uses RR process \((\text{arrived\_time, duration})\)
- \(P_1(0,20), P_2(30,10), P_3(20,40), P_4(40,25)\)
- Time quantum 15

Which of the following is the correct total \textit{waiting time} of the above processes?

A. 40
B. 50
C. 60
D. 70
Inter-process Communication (IPC)
Objectives

- Present what IPC is
- Present why we need synchronization
  - Methods of synchronization
  - Classical synchronization problems
- Write a simple synchronization program
Reference

- Chapter 3, 6 of *Operating System Concepts*
Introduction

- In some situations, processes need to communicate with each other
  - To send/receive data (web browser – web server)
  - To control the other process
  - To synchronize with each other
- This can be done by IPC
- IPC is implemented differently among OSes
  - Linux: message queue, semaphore, shared segment, …
Introduction (cont’d)

- IPC can be divided into 2 categories
  - IPC among processes within the same system
    - Linux: pipe, named pipe, file mapping, …
  - IPC among processes in different systems
    - Remote Procedure Call (RPC), Socket, Remote Method Invocation (RMI), …
Process Synchronization
Synchronization definition

- **Process synchronization** refers to the idea that multiple processes are to join up or handshake at a certain point, in order to reach an agreement or commit to a certain sequence of actions.
Synchronization is everywhere
Synchronization is everywhere

Producer  Work Queue  Worker

another example ...
Synchronization is everywhere
Synchronization is everywhere
Synchronization is everywhere

another example ...
Synchronization is everywhere

Team Work
Problem

Write process $P$:

```java
while (true) {
    val = buf;
    val += count(); // Take time
    buf = val;
}

buf: Buffer

UPDATE A SET
    buf = buf + count();
```
Problem

Write process $P$:

```java
while (true) {
    val = buf;
    val += count(); // Take time
    buf = val;
}
```

buf: Buffer

UPDATE A SET
    buf = buf + count();

What if more than one $P$ are running?
Problem (cont’d)

- Two concurrent processes

\[
\begin{align*}
\text{val} &= \text{buf}; \\
\text{val} &= \text{buf}; \\
\text{val} &= \text{count}(); \\
\text{val} &= \text{count}(); \\
\text{buf} &= \text{val}; \\
\text{buf} &= \text{val};
\end{align*}
\]

Do we always get the expected value of \texttt{buf}? Why?
Suppose buf=5

val=buf; // val=5
val+=count(); // val=10
val=buf; // val=5
val+=count(); // val=10
buf=val; // buf=10
buf=val; // buf=10
Problem (cont’d)

- **Cause:** $P$ and $Q$ simultaneously operate on global variable `buf`
- **Solution:** Let them operate **separately**

```plaintext
val = buf;       // val = 5
val += count(); // val = 10
buf = val;      // buf = 10

val = buf;      // val = 10
val += count(); // val = 15
buf = val;      // buf = 15
```
Race condition

- Happen when many processes simultaneously work with shared data
Race condition

- Happen when many processes simultaneously work with shared data
Race condition

- Happen when many processes simultaneously work with shared data
Race condition

- Happen when many processes simultaneously work with shared data
Race condition

- Happen when many processes simultaneously work with shared data
Race condition

• Happen when many processes simultaneously work with shared data
Race condition

- Happen when many processes simultaneously work with shared data

To avoid “trouble”, processes need to be controlled
Critical section

- In **concurrent programming** a **critical section** is a piece of **code** that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one **thread of execution**. A critical section will usually terminate in fixed time, and a thread, task or process will have to wait a fixed time to enter it (aka bounded waiting). Some **synchronization** mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a **semaphore**.

Critical section

- Suppose $n$ processes $P_1, ..., P_n$ share a global variable $v$
  - $v$ can also be other resource, e.g, file
- Each process has a segment of code $CS_i$ which operates on $v$
  - $CS_i$ is called critical section
  - Because it is critical to prone errors
  - $CS_i$ should be the smallest code segment
- Need to make the critical section safe
Critical section
Critical section

Train can pass if semaphore is UP

Semaphore (up position)

Leave trigger

Pass trigger

Semaphore (up position)

Leave trigger

Never allow more than one train on this part of the track !!!
Process $P$: 
while (true) {
    waitForNewRequest();
    if(found){
        hit+=1;
        val=hit;
    }
    Respond();
} 
hit: a global variable

Which is the critical section of the code when multiple processes of $P$ run?
Question

Process $P$:
while (true) {
        waitForNewRequest();
        if(found){
            hit+=1;
            val=hit;
        }
        Respond();
    }
hit: a global variable

Which is the critical section of the code when multiple processes of $P$ run?

While (true) {
        waitForNewRequest();
        if(found){
            hit+=1;
            val=hit;
        }
        Respond();
    }
Critical section (cont’d)

- Common structure

do {
    Enter_Section (CS_i);
    Run CS_i;
    Exit_Section(CS_i);
    Run (REMAIN_i); // Remainder section
} while (TRUE);
Critical section (cont’d)

- Short description

do {
    ENTRY$_i$; // Enter section
    Run CS$_i$; // Critical section
    EXIT$_i$; // Exit section
    REMAIN$_i$; // Remainder section
} while (TRUE);
Implementation of Critical section

Implementation must satisfy 3 conditions

1. **Mutual Exclusion**
   - If a process is in its critical section, then no other processes can be in their critical sections

2. **Progress**
   - If no process is in its critical section
   - other processes waiting to enter their critical section,
   - then the selection of the process to enter the critical section cannot be postponed indefinitely

3. **Bounded Waiting**
   - No process has to wait indefinitely to enter its critical section
Question

Which is the purpose of the first condition?

A. It supports the priority of process
B. It ensures the correct use of the shared resource
C. It tries to utilize the shared resource effectively
D. It makes the implementation of OS simpler
Critical section

Mutual exclusion using critical regions
Question

Which is the consequence of the second condition?

A. It reduces the waiting time of requested processes
B. It ensures the correct use of the shared resource
C. It supports the priority of processes
D. It makes the implementation of OS simpler
Question

Which is the consequence of the second condition?

A. It supports the priority of processes
B. It ensures the correct use of the shared resource
C. It utilizes the shared resource effectively
D. It makes the algorithm complicated to implement
Question
Which is the consequence of the 3\textsuperscript{rd} condition?

A. It supports the priority of processes
B. It ensures the correct use of the shared resource
C. It utilizes the shared resource effectively
D. It makes sure no process can never enter its critical section
Question

Which is the correct conditions of critical section?

A. mutual exclusion, protection, bounded using
B. mutual exclusion, protection, bounded waiting
C. mutual exclusion, progressive, bounded waiting
D. mutual exclusion, bounded waiting, progress
Question

Which is the correct purpose the 2\textsuperscript{nd} condition of critical section?

A. maximize CPU utilization
B. maximize the shared resource utilization
C. maximize disk utilization
D. maximize RAM utilization
Question

Which is the consequence of the 3\textsuperscript{rd} condition?

A. It supports the priority of processes
B. It ensures the correct use of the shared resource
C. It ensures the relative fairness of processes to use the shared resource
D. It utilizes the shared resource effectively
The fairness

The fair exam today is to swim
Critical section (cont’d)

- Each process has to
  - request to run (enter section) its critical section $CS_i$
  - and announce its completion (exit section) of its $CS_i$. 
Peterson’s Solution

- Solution for two processes
- The two processes share two variables:
  - `int turn; // with the value of 0 or 1`
  - `Boolean flag[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
  - If `turn==i` then $P_i$ is in turn to run its $CS_i$
- 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!
Peterson’s solution (cont’d)

- Program $P_i$
  do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j) ;
    CS$_i$;
    flag[i] = FALSE;
    REMAIN$_i$;
  } while (1);
Peterson’s solution (cont’d)

- The proof of this solution is provided on page 196 of the textbook
- Comments
  - Complicated when the number of processes increases
  - Difficult to control
Question

Which code snippet is Enter_Section?

A. \[
    \text{flag}[i] = \text{TRUE}; \\
    \text{turn} = j; \\
    \text{while} \ (\text{flag}[j] \&\& \text{turn} == j) ;
\]

B. \[
    \text{flag}[i] = \text{TRUE}; \\
    \text{while} \ (\text{flag}[j] \&\& \text{turn} == j) ;
\]

C. \[
    \text{flag}[i] = \text{TRUE}; \\
    \text{turn} = j; \\
\]

D. \[
    \text{turn} = j; \\
    \text{while} \ (\text{flag}[j] \&\& \text{turn} == j) ;
\]
Semaphore
Reference information

- Semaphore is proposed by Edsger Wybe Dijkstra (Dutch) for Computer Science in 1972
- Semaphore was firstly used in his book “The operating system”

Edsger Wybe Dijkstra (1930-2002)
Semaphore

- Semaphore is an integer, can be only access through two atomic operators wait (or P) and signal (or V).
  - P: proberen – check (in Dutch)
  - V: verhogen – increase (in Dutch)
- Processes can share a semaphore
- Atomic operators guarantee the consistency
wait and signal operators

wait(S)  // or P(S)
{
    while (S<=0);
    S--;
}

- Wait if semaphore S<=0 else decrease S by 1

signal(S)  // or V(S)
{
    S++;
}

- Increase S by 1
Using semaphore

- Apply for critical section

```c
do {
    wait(s); // s is a semaphore initialized by 1
    CS_j;
    signal(s);
    REMAIN_j;
} while (1);
```
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Semaphore
Question

Process $P$:
while (true) {
    waitForNewRequest();
    if(found){
        hit+=2;
        val=hit;
    }
    Respond();
}

hit: a global variable

Use semaphore to make the code safe?
**Question**

**Process P:**

```java
while (true) {
    waitForNewRequest();
    if (found) {
        hit += 2;
        val = hit;
    }
    Respond();
}
```

*hit: a global variable*

Use semaphore to make the code safe?

```java
if (found) {
    wait(mutex);
    hit += 2;
    val = hit;
    signal(mutex);
}
```
Problem (cont’d)

- Two concurrent processes: mutex=1

```c
wait(mutex)
val=buf;
val += count();
buf=val;
signal(mutex);
```

```c
wait(mutex)

val=buf;
val += count();
buf=val;
signal(mutex);
```

(waiting)
Using semaphore (cont’d)

- $P_1$ needs to do $O_1$; $P_2$ need to do $O_2$; $O_2$ can only be done after $O_1$
- Solution: use a semaphore $synch = 0$
Using semaphore (cont’d)

- $P_1$ needs to do $O_1$; $P_2$ need to do $O_2$; $O_2$ can only be done after $O_1$
- Solution: use a semaphore $synch = 0$

- $P_1$:
  ...  
  $O_1$;  
  signal(synch);  
  ...  

- $P_2$:
  ...  
  wait(synch);  
  $O_2$;  
  ...
Using semaphore (cont’d)

- $P_1$ needs to do $O_1$; $P_2$ need to do $O_2$; $O_2$ can only be done after $O_1$
- Solution: use a semaphore $synch = 0$

- $P_1$:
  ...
  $O_1$;
  signal(synch);
  ...

- $P_2$:
  ...
  wait(synch);
  $O_2$;
  ...

Semaphore support

To release or signal a semaphore, we use the `sem_post` function:

```c
int sem_post(sem_t *sem);
```

A semaphore is initialised by using `sem_init` (for processes or threads) or `sem_open` (for IPC).

```c
sem_init(sem_t *sem, int pshared, unsigned int value);
```

Where,

- **sem**: Specifies the semaphore to be initialized.
- **pshared**: This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.
- **value**: Specifies the value to assign to the newly initialized semaphore.

To destroy a semaphore, we can use `sem_destroy`.

```c
sem_destroy(sem_t *mutex);
```

To declare a semaphore, the data type is `sem_t`.

https://www.geeksforgeeks.org/use-posix-semaphores-c/
Semaphore support

Semaphore, Shared Memory and IPC

- Introduction
- Installing/Configuring
  - Requirements
  - Installation
  - Runtime Configuration
  - Resource Types

- Predefined Constants
- Semaphore Functions
  - ftok — Convert a pathname and a project identifier to a System V IPC key
  - msg_get_queue — Create or attach to a message queue
  - msg_queue_exists — Check whether a message queue exists
  - msg_receive — Receive a message from a message queue
  - msg_remove_queue — Destroy a message queue
  - msg_send — Send a message to a message queue
  - msg_set_queue — Set information in the message queue data structure
  - msg_stat_queue — Returns information from the message queue data structure
  - sem_acquire — Acquire a semaphore
  - sem_get — Get a semaphore id
  - sem_release — Release a semaphore
  - sem_remove — Remove a semaphore
  - shm_attach — Creates or open a shared memory segment
  - shm_detach — Disconnects from shared memory segment
  - shm_get_var — Returns a variable from shared memory
  - shm_has_var — Check whether a specific entry exists
  - shm_put_var — Inserts or updates a variable in shared memory

Semaphore support

In the following example, we will implement a simple login queue to limit the number of users in the system:

```java
class LoginQueueUsingSemaphore {
    private Semaphore semaphore;

    public LoginQueueUsingSemaphore(int slotLimit) {
        semaphore = new Semaphore(slotLimit);
    }

    boolean tryLogin() {
        return semaphore.tryAcquire();
    }

    void logout() {
        semaphore.release();
    }

    int availableSlots() {
        return semaphore.availablePermits();
    }
}
```

Notice how we used the following methods:

- `tryAcquire()` - return true if a permit is available immediately and acquire it otherwise return false, but `acquire()` acquires a permit and blocking until one is available
- `release()` - release a permit
- `availablePermits()` - return number of current permits available

https://www.baeldung.com/java-semaphore
Semaphore support

```python
class asyncio.Semaphore(value=1, *, loop=None)
A Semaphore object. Not thread-safe.

A semaphore manages an internal counter which is decremented by each acquire() call and incremented by each release() call. The counter can never go below zero; when acquire() finds that it is zero, it blocks, waiting until some task calls release().

The optional value argument gives the initial value for the internal counter (1 by default). If the given value is less than 0 a ValueError is raised.

Deprecated since version 3.8, will be removed in version 3.10: The loop parameter.

The preferred way to use a Semaphore is an async with statement:

```python
sem = asyncio.Semaphore(10)
# ... later
async with sem:
    # work with shared resource
```

which is equivalent to:

```python
sem = asyncio.Semaphore(10)
# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```

coroutine acquire()

Acquire a semaphore.

If the internal counter is greater than zero, decrement it by one and return True immediately. If it is zero, wait until a release() is called and return True.

locked()

Returns True if semaphore can not be acquired immediately.

release()

Release a semaphore, incrementing the internal counter by one. Can wake up a task waiting to acquire the semaphore.
Semaphore support

package semaphore

import "golang.org/x/sync/semaphore"

Package semaphore provides a weighted semaphore implementation.

Example (WorkerPool)

Index

type Weighted
- func NewWeighted(n int64) *Weighted
- func (s *Weighted) Acquire(ctx context.Context, n int64) error
- func (s *Weighted) Release(n int64)

Examples

package (WorkerPool)

Package Files

semaphore.go
Semaphore support

```csharp
private static void Worker(object num)
{
    // Each worker thread begins by requesting the
    // semaphore.
    Console.WriteLine("Thread \{0\} begins \" +
    "and waits for the semaphore.\", num);
    pool.WaitOne();

    // A padding interval to make the output more orderly.
    int padding = Interlocked.Add(ref _padding, 100);

    Console.WriteLine("Thread \{0\} enters the semaphore.\", num);

    // The thread's "work" consists of sleeping for
    // about a second. Each thread "works" a little
    // longer, just to make the output more orderly.
    //
    //Thread.Sleep(1000 + padding);

    Console.WriteLine("Thread \{0\} releases the semaphore.\", num);
    pool.Release();
}
```
Semaphore support

### Remarks

Semaphore support

Semaphores are useful in controlling access to a shared resource that can only support a limited number of users. The current count of the CSemaphore object is the number of additional users allowed. When the count reaches zero, all attempts to use the resource controlled by the CSemaphore object will be inserted into a system queue and wait until they either time out or the count rises above 0. The maximum number of users who can access the controlled resource at one time is specified during construction of the CSemaphore object.

To use a CSemaphore object, construct the CSemaphore object when it is needed. Specify the name of the semaphore you wish to wait on, and that your application should initially own it. You can then access the semaphore when the constructor returns. Call CSyncObject::Unlock when you are done accessing the controlled resource.

An alternative method for using CSemaphore objects is to add a variable of type CSemaphore as a data member to the class you wish to control. During construction of the controlled object, call the constructor of the CSemaphore data member specifying the initial access count, maximum access count, name of the semaphore (if it will be used across process boundaries), and desired security attributes.

To access resources controlled by CSemaphore objects in this manner, first create a variable of either type CSingleLock or type CMultiLock in your resource's access member function. Then call the lock object's Lock member function (for example, CSingleLock::Lock). At this point, your thread will either gain access to the resource, wait for the resource to be released and gain access, or wait for the resource to be released and time out, failing to gain access to the resource. In any case, your resource has been accessed in a thread-safe manner. To release the resource, use the lock object's Unlock member function (for example, CSingleLock::Unlock) or allow the lock object to fall out of scope.

Alternatively, you can create a CSemaphore object stand-alone, and access it explicitly before attempting to access the controlled resource. This method, while clearer to someone reading your source code, is more prone to error.

For more information on how to use CSemaphore objects, see the article Multithreading: How to Use the Synchronization Classes.
Semaphore support

**semaphore.js**

Install: npm install semaphore

Limit simultaneous access to a resource.

```javascript
// Create
var sem = require('semaphore')(capacity);

// Take
sem.take(fn[, n=1])
sem.take(n, fn)

// Leave
sem.leave([n])

// Available
sem.available([n])
```
Semaphore implementation

- In the above semaphore implementation
  - Use **busy waiting** (while loop)
  - Resource wasting
- Atomic operators
  - When a process called `wait()`, it will be **blocked** if the semaphore is not free
    - This type of semaphore is called **spinlock**
  - Other `wait()` implementation just returns true/false and **does not block** the calling process
Semaphore implementation (cont’d)

- Remove the busy waiting loop by using `block`
- To restored a blocked process, use `wakeup`
- Semaphore data structure

```c
typedef struct {
    int value; // value of semaphore
    struct process *L; // waiting process list
} semaphore;
```
Semaphore implementation (cont’d)

```c
void wait(semaphore *S) {
    S->value--;  
    if (S->value<0) {
        Add the requested process $P$ into S->L;  
        block($P$);  
    }
}
```

```c
void signal(semaphore *S) {
    S->value++;  
    if (S->value<=0) {
        remove a process $P$ from S->L;  
        wakeup($P$);  
    }
}
```
Semaphore implementation (cont’d)
Semaphore implementation (cont’d)
Binary semaphore

- Semaphore only has the value of 0 or 1
- Other semaphore type is **counting semaphore**
Binary semaphore

- Semaphore only has the value of 0 or 1
- Other semaphore type is counting semaphore
Binary semaphore

- Semaphore only has the value of 0 or 1
- Other semaphore type is counting semaphore
Question

- When counting semaphores are suitable to use?
  A. When 2 processes share a single variable/resource
  B. When 3 processes share a single variable/resource
  C. When $n$ processes share a single variable/resource
  D. When $n$ processes share $m$ variables/resources of the same type
Classical synchronization problems
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’d)

<table>
<thead>
<tr>
<th>Write process $P$:</th>
<th>Read process $Q$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>do {</td>
<td>do {</td>
</tr>
<tr>
<td>wait(empty);</td>
<td>wait(full);</td>
</tr>
<tr>
<td>wait(mutex);</td>
<td>wait(mutex);</td>
</tr>
<tr>
<td>Write(item,buf);</td>
<td>Read(item,buf);</td>
</tr>
<tr>
<td>signal(mutex);</td>
<td>signal(mutex);</td>
</tr>
<tr>
<td>signal(full);</td>
<td>signal(empty);</td>
</tr>
<tr>
<td>} while (TRUE);</td>
<td>} while (TRUE);</td>
</tr>
</tbody>
</table>

buf: shared resource
Question

Which is the initialized value of the *full* variable in the above algorithm?

A. -1
B. 0
C. 1
D. NULL
Question

What will be the problem if the initialized value of the *full* variable is 1?

A. no problem at all
B. the writer process can not run
C. the reader process can not run
D. the reader can read an invalid value
Bounded-buffer problem (cont’d)
Bounded-buffer problem (cont’d)
Readers-writers problem

- A data set is shared among a number of concurrent processes
  - Readers – only read
  - Writers – can both read and write

Problem

- allow multiple readers to read at the same time when there is no writer accessing the data set
- Only one writer can access the shared data at the a time
Readers-writers problem
Readers-writers problem (cont’d)

- Shared data
  - Data set
  - Semaphore \textit{wrt} initialized by 1
    - Used to manage \textit{write} access
  - Integer \textit{readcount} initialized by 0 to count the number of readers that are \textit{reading}
  - Semaphore \textit{mutex} initialized by 1
    - Used to manage \textit{readcount} access
Readers-writers problem (cont’d)

- Process writer $P_w$:
  
do {
    wait(wrt);
    write(data_set);
    signal(wrt);
  }while (TRUE);

- Process reader $P_r$:
  
do {
    wait(mutex);
    readcount++;
    if (readcount == 1) wait(wrt);
    signal(mutex);
    read(data_set);
    wait(mutex);
    readcount--;
    if (readcount == 0) signal(wrt);
    signal(mutex);
  }while (TRUE);
Question

Why do we need readcount variable?

A. We may remove this variable
B. To make sure there is one reader at a time
C. To make sure no readers are reading
D. To make sure no readers are reading before writing
Question

Which is the initialized value of the readcount variable in the above algorithm?

A. -1
B. 0
C. 1
D. NULL
Question

Which is the purpose of *mutex* variable?

A. To safely access the `data_set`
B. We may remove this variable without affecting the program
C. To safely access the `readcount` variable
D. To safely access the `wrt` variable
Question

Which is the initialized value of the mutex variable in the above algorithm?

A. -1
B. 0
C. 1
D. NULL
Question

Which is the purpose of \textit{wrt} variable?

A. To safely access the \textit{mutex} variable
B. To safely write the \textit{data\_set}
C. To safely write the \textit{readcount} variable
D. To safely read the \textit{data\_set}
Question

Which is the initialized value of the *wrt* variable in the above algorithm?

A. -1  
B. 0  
C. 1  
D. NULL
Dining-Philosophers Problem

- Five philosophers at a table having 5 chopsticks, 5 bows and a rice cooker
- A philosopher just eats or thinks
- How to make sure philosophers correctly use the “shared data” – the chopsticks
Dining-philosophers problem (cont’d)

- Use semaphore to handle chopstick access
  - semaphore chopstick[5];
- Solution is provided as in the text box

Code of philosopher $i$:

```c
do {
  wait(chopstick[i]);
  wait(chopstick[(i+1)%5];
  Eat(i);
  signal(chopstick[i]);
  signal(chopstick[(i+1)%5];
  Think(i);
} while (TRUE);
```
Question

- What value chopstick[i] is initialized?
  A. 1
  B. 2
  C. 0
  D. 5
Question

Is there any problem with the solution?

A. No problem
B. Only one philosopher can eat at a time
C. Only three philosophers can eat at a time
D. No philosopher could eat in case each takes a chopstick and waits for the second one
Question

Which of the following is incorrect about the solution to the above problem?

A. No solution available
B. Create an order of philosophers to eat
C. Create an order of philosophers to think
D. Allow at most 4 philosophers to request to eat at a time
Extra problem: Barrier

- Use of a barrier
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through
Limitations of semaphore (cont’d)

- Compare the two code snippets

**Snippet 1**

...  
wait(mutex);  
//Critical section  
signal(mutex);  
...  

**Snippet 2**

...  
signal(mutex);  
//Critical section  
wait(mutex);  
...
Question

What is the problem of the two code snippets?

A. Snippet 1 has problem
B. Snippet 2 has problem
C. Both snippets have problem
D. No problem at all
Question

- Which is the problem of the incorrect use of semaphore in the above code snippet?
  A. No process can enter its critical section
  B. No problem at all
  C. The mutual exclusion condition may be violated
  D. No process can exit its critical section
Limitations of semaphore

- Semaphores need correct calls to \textit{wait} and \textit{signal}
- Incorrect use of semaphore may lead to deadlock
- Even \textbf{correct use} of semaphores may lead to deadlock, in some cases
Limitations of semaphore (cont’d)

- Compare the two code snippets

<table>
<thead>
<tr>
<th>Snippet 1</th>
<th>Snippet 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>wait(mutex);</td>
<td>wait(mutex);</td>
</tr>
<tr>
<td>CS₁;</td>
<td>CS₂;</td>
</tr>
<tr>
<td>wait(mutex);</td>
<td>signal(mutex);</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Question

Which of the two code snippets has problem?
A. Snippet 1 has problem
B. Snippet 2 has problem
C. Both snippets have problem
D. No problem at all
Question

Which is the consequence of the above problem?
A. One process will be blocked
B. There will be a deadlock
C. No consequences if only two processes are involved
D. No consequences
Limitations of semaphore (cont’d)

- Process $P_1$
  
  ...  
  wait(S);
  wait(Q);
  CS1...
  signal(S);
  signal(Q);
  ...

- Process $P_2$
  
  ...  
  wait(Q);
  wait(S);
  CS2...
  signal(Q);
  signal(S);
  ...
Question

What is the problem of the above two processes?

A. There is deadlock
   - if $P_1$ got S and waits for Q and
   - $P_2$ got Q and waits for S
B. The exclusive condition is violated
C. The order of semaphore calls is incorrect
D. No problem at all
Monitor
Reference information

- Per Brinch Hansen (Dennish) proposed the concept and implemented in 1972
- Monitor was firstly used in Concurrent Pascal programming language

Per Brinch Hansen (1938-2007)
What is monitor?

- Monitor means to **supervise**
- It is a type of construct in a high level programming language for synchronization purpose
  - C# programming language
  - Java programming language
    - [http://journals.ecs.soton.ac.uk/java/tutorial/java/threads/monitors.html](http://journals.ecs.soton.ac.uk/java/tutorial/java/threads/monitors.html)
    - [http://www.csc.villanova.edu/~mdamian/threads/javamonitors.html](http://www.csc.villanova.edu/~mdamian/threads/javamonitors.html)
- Monitor was studied and developed to overcome the limitations of semaphores
C# monitor

The following is the syntax for using a monitor.

```csharp
try {
    int x = 1;
    Monitor.Enter(x);

    try {
        // Code that needs to be protected by the monitor.
    } finally {
        Monitor.Exit(x);
    }
}

catch (SynchronizationLockException SyncEx) {
    Console.WriteLine("A SynchronizationLockException occurred. Message:");
    Console.WriteLine(SyncEx.Message);
}
```
Java monitor

Figure 20-1: A Java monitor.
Monitor

- A monitor usually has
  - Member variables as shared resources
  - A set of procedures which operate on the shared resources
  - Exclusive lock
  - Constraints to manage race condition
- This description of monitor is like a class
A sample monitor type

monitor monitor_name {
    //Shared resources
    procedure P1(...) { ... }
    procedure P2(...) { ... }
    ...
    procedure Pn(...) { ... }
    initialization_code (..) { ... }
}
Schematic view of a Monitor
Monitor implementation

- Monitor must be implemented so that
  - only one process can enter the monitor at a time (mutual exclusive)
  - programmer do not need to write code for this
- Other monitor implementation
  - have more synchronization mechanism
  - add condition variable
Condition type

- Declaration
  - condition x, y;

- Use condition variable
  - there are two operators: wait and signal
  - x.wait():
    - process calls x.wait() will have to wait or suspend
  - x.signal():
    - process calls x.signal() will wakeup a waiting process – the one that called x.wait()
Monitor with condition

Queues associated with x, y conditions

Shared data

Operations

Initialization code

Entry queue
**x.signal() characteristics**

- `x.signal()` wakeup only one waiting process
- If no waiting process, it does nothing
- `x.signal()` is different from that of classical semaphore
  - Signal in classical semaphore always change the state (value) of semaphore
Solution to Dining Philosophers

monitor DP
{
    enum { THINKING, HUNGRY, EATING } state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        test((i + 4) % 5);
        if (state[i] != EATING) self [i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}
Solution to Dining Philosophers (cont)

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

initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
```
Solution to Dining Philosophers (cont)

● Each philosopher invokes the operations `pickup()` and `putdown()` in the following sequence

```
dp.pickup (i)
EAT
dp.putdown (i)
```
Monitor Implementation Using Semaphores

- Variables
  
  ```
  semaphore mutex;  // (initially  = 1)
  semaphore next;   // (initially  = 0)
  int next-count = 0;
  ```

- Each procedure $F$ will be replaced by

  ```
  wait(mutex);
  ...
  //body of $F$;
  ...
  if (next-count > 0)
      signal(next)
  else
      signal(mutex);
  ```

- Mutual exclusion within a monitor is ensured.
Monitor Implementation

- For each condition variable $x$, we have:

  semaphore $x$-sem; // (initially $= 0$)
  int $x$-count = 0;

- The operation $x$.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:

```c
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}
```
Linux Synchronization

- **Linux:**
  - disables interrupts to implement short critical sections

- **Linux provides:**
  - semaphores
  - spin locks
Pthreads Synchronization

- pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks
Question?