Synchronization, Part II

CSD2180 Operating Systems
BSc in Computer Science (IMGD / RTIS)
Singapore Institute of Technology / DigiPen Institute of Technology
September 2021

Attendance Taking

https://forms.office.com/r/7zRQtKQu9n

- Log in to your SIT account to submit the form
- You can only submit the form once
- The codeword is **green**

Previously On...

The Producer-Consumer Problem

Suppose at one point, `counter` == 5

Concurrently

- Producer produces one item
- Consumer consumes one item

We should now have `counter` == 5

But we may end up with `counter` == 4, 5, or 6! (why?)

```
/* le producer */
while (true) {
    /* produce an item in next_produced */
    while (counter == BUFFER_SIZE); /* do nothing */

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

```
/* le consumer */
while (true) {
    while (counter == 0); /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;

    /* consume the item in next consumed */
}
```

An Attempt at a Software Solution

```
/* at P0 */
while (true) {

    /* entry section */
    while (turn == (1 - i));
    . . .
    /* critical section */
    . . .
    /* exit section */
    turn = (1 - i);
    . . .

    /* remainder section done quickly */
    /* back to start of while(true) */
}
```

```
/* at P1 */
while (true) {

    /* entry section */
    while (turn == (1 - i));
    . . .
    /* critical section */
    . . .
    /* exit section */
    turn = (1 - i);
    . . .

    /* okay, i'm gonna chill */
    /* proceeds to run indefinitely... */
}
```

An Attempt at a Software Solution

```
/* at P0 */
while (true) {

    /* entry secti
    while (turn == (1 - i));
    . . .
    /* critical section */
    . . .
    /* exit section */
    turn = (1 - i);
    . . .

    /* remainder section done quickly */
    /* back to start of while(true) */
}
```

Dude, can I enter my critical section now? 🤔

```
/* at P1 */
while (true) {

    /* entry section */
    while (turn == (1 - i));
    . . .
    /* critical section */
    . . .
    /* exit section */
    turn = (1 - i);
    . . .

    /* okay, i'm
    /* proceeds to run indefinitely... */
}
```

Hahaha... No ☺

Peterson's Solution Example

```
while (true) {

    /* entry section */
    flag[i] = true;
    turn = (1 - i);
    while (turn == (1 - i) && flag[1 - i]);

    . . .
    /* critical section */
    . . .

    /* exit section */
    flag[i] = false;

    . . .

    /* remainder section done quickly */
    /* back to start of while(true) */
}
```

```
while (true) {

    /* entry section */
    flag[i] = true;
    turn = (1 - i);
    while (turn == (1 - i) && flag[1 - i]);

    . . .
    /* critical section */
    . . .

    /* exit section */
    flag[i] = false;

    . . .

    /* okay, i'm gonna chill */
    /* proceeds to run indefinitely... */
}
```

Peterson's Solution Example

```
while (true) {

    /* entry sectio
    flag[i] = true;
    turn = (1 - i);
    while (turn == (1 - i) && flag[1 - i]);

    . . .
    /* critical section */
    . . .

    /* exit section */
    flag[i] = false;

    . . .

    /* remainder section done quickly */
    /* back to start of while(true) */
}
```

Dude, can I enter my critical section now? 🤔

```
while (true) {

    /* entry section */
    flag[i] = true;
    turn = (1 - i);
    while (turn == (1 - i) && flag[1 - i]);

    . . .
    /* critical section */
    . . .

    /* exit section */
    flag[i] = false;

    . . .

    /* okay, i'm g
    /* proceeds to run indefinitely... */
}
```

Okay, sure! ☺

Although useful for demonstration, Peterson's Solution is not guaranteed to work on modern architectures... 🍔

And Now the Conclusion…

Why? 🍔

Peterson's Solution and Modern Architectures

Suppose two threads share

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

What is the expected output? `100`

However, since `flag` and `x` are independent, the instructions in Thread 2 may be executed *out-of-order*, i.e., dynamic execution

We may get 0 as output

```
/* Thread 1 */
while (!flag);
print x;
```

```
/* Thread 2 (by right) */
x = 100;
flag = true;
```

Peterson's Solution and Modern Architectures

```
/* flag == false; x == 0 */

/* Thread 1 */
while (!flag);
print x;
```

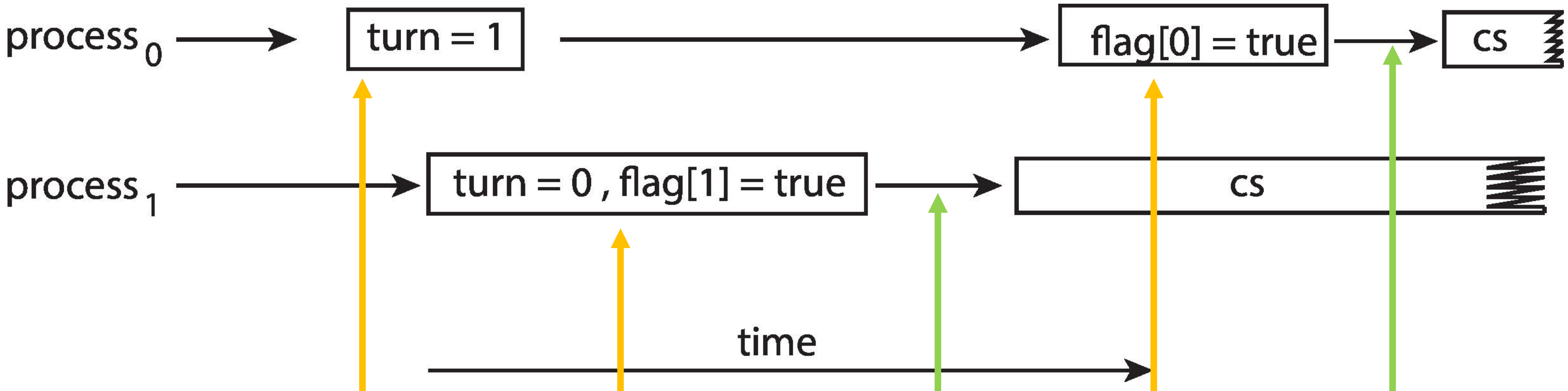
```
/* flag == false; x == 0 */

/* Thread 2 (by right) */
x = 100;
flag = true;
```

2. Since `flag == true`, exit loop
3. Prints x, which is `x == 0`

1. Since `flag` and `x` are independent, CPU first executes `flag = true`
4. Sets `x = 100`

Peterson's Solution and Modern Architectures



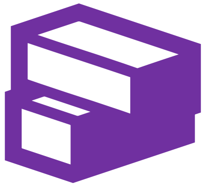

```
                                                                        /* enter */                      /* enter */

turn == 1; flag[0] == false; flag[1] == false;                                   turn == 0; flag[0] == true; flag[1] == true;

                           turn == 0; flag[0] == false; flag[1] == true;

(turn == i || !flag[1 - i]) /* condition to enter critical section: my turn, or the other process is not ready */
```

To ensure that Peterson's Solution work correctly on modern computer architectures, we can use memory barriers! 🚧

Wait, what? 

Hardware Support for Synchronization

Memory Barrier

Instruction that forces any change in memory to be propagated to all other processors

When instruction is performed, system ensures that all loads and stores are completed before subsequent loads or stores are performed

In the previous example, can add a memory barrier to ensure Thread 1 outputs 100

- **Thread 1:** The value of `flag` will be loaded before the value of x
- **Thread 2:** The assignment to `x` will occur before the assignment to `flag`

```
/* Thread 1 */
while (!flag) memory_barrier();
print x;
```

```
/* Thread 2 */
x = 100;
memory_barrier();
flag = true;
```

How do we implement memory barriers in Peterson's Solution?

Hardware Instructions

Many systems provide hardware support for implementing critical section code

In uniprocessor systems, can just disable interrupts

- Running code executes without preemption
- Generally, too inefficient on multiprocessor systems
- **Operating systems using this approach not broadly scalable**

Many modern computer systems provide special hardware instructions that atomically

- ***Test and modify*** **the content of a word (`test_and_set()`)**
- ***Swap the contents*** **of two words (`compare_and_swap()`)**

test_and_set()

```
/* test-and-set instruction */
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
```

Executed atomically

Returns the original value of passed parameter

Set the new value of passed parameter to `true`

```
/* mutual exclusion using test-and-set */
while (true) {

    /* entry section */
    /* shared boolean variable lock */
    while (test_and_set(&lock));

    . . .
    /* critical section */
    . . .

    /* exit section */
    lock = false;
    . . .

    /* remainder section */
}
```

compare_and_swap()

```
/* compare-and-swap */
int compare_and_swap(int *val, int exp, int new_val) {
    int temp = *val;
    if (*val == exp)
        *val = new_val;
    return temp;
}
```

Executed atomically

Returns the original value of passed parameter `val`

Set the variable `val` to the value of the parameter `new_val`, but only if `*val == exp` is true

```
/* mutual exclusion using compare-and-swap */
while (true) {

    /* entry section */
    /* shared boolean value lock */
    while (compare_and_swap(&lock, 0, 1) != 0);

    . . .
    /* critical section */
    . . .

    /* exit section */
    lock = 0;
    . . .

    /* remainder section */
}
```

Do the previous examples satisfy progress?

Do they satisfy bounded waiting? 📦

Mutex Locks

Mutex, short for *mut*ual *ex*clusion

Mutex Locks

Previous solutions generally inaccessible to application programmers

Hence operating system designers build software approach to solve critical section problem

Simplest of these is mutex lock

Implemented as a boolean variable indicating if lock is available or not

```
while (true) {

    /* acquire lock */
    . . .
    /* critical section */
    . . .
    /* release lock */
    . . .

    /* remainder section */
}
```

Protects a critical section by having a process

- `acquire()` a lock when entering critical section
- `release()` the lock when exiting critical section

Mutex Locks

Calls to `acquire()` and `release()` must be atomic

- e.g., Can be implemented via hardware atomic instructions such as compare-and-swap

The implementation here requires **busy waiting**

- Processes loop continuously in the call to `acquire()`
- Wastes CPU cycles, but no context switching required; hence a tradeoff
- This lock therefore called a **spinlock**, i.e., process *spins* while waiting for lock to become available

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

```
/* release the lock 'available' */
release() {
    available = true;
}
```

Semaphores

Semaphores

Semaphore `S` is an *integer* variable that can only be accessed via two atomic operations `wait()` and `signal()`

How it works

- `S` is initialized to the number of resources available
- Each process that wishes to use a resource performs a `wait()` operation on the semaphore
- When a process releases a resource, it performs a `signal()` operation
- When `S == 0`, all resources are being used; processes that wish to use a resource will block until `S > 0`

Protip: You may encounter the use of `P()` and `V()` in place of `wait()` and `signal()`

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

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

Semaphores

Two types of semaphores

- **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 implement a binary semaphore from a counting semaphore (how?)

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

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

Semaphore Usage

Use case 1: Solution to the critical section problem

- Create a semaphore `mutex` initialized to 1

```
wait(mutex);

/* critical section */

signal(mutex);
```

Use case 2: Two concurrently running processes, P1 with statement S1, and P2 with statement S2; S2 must only execute after S1 has completed

- Can enforce this by having P1 and P2 share a common semaphore `synch` initialized to 0

```
/* at P1 */
S1;
signal(synch);

/* at P2 */
wait(synch);
S2;
```

Semaphore Implementation

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

This means that `wait()` and `signal()` now becomes a critical section problem

- In previous implementation, can have busy waiting in critical section

Can we do better?

Semaphore implementation without busy waiting

- With each semaphore `S` there is an associated *waiting queue*
- When a process executes `wait()` and finds the value of `S <= 0`, it suspend itself into the waiting queue
- A process that is suspended and waiting on a semaphore should be restarted when some other process executes a `signal()` operation
- The process is restarted by a `wakeup()` operation, which changes the process from the waiting state to the ready state

Semaphore Implementation without Busy Waiting

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

```c
/* semaphore wait() */
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        /* adds this process to S->list */
        sleep();
    }
}
```

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

sleep(): 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 without Busy Waiting

T+0:	Semaphore `S->value` initialized to 1	`S->{value == 1; list == {}}`
	P1, P2, P3 arrives	`S->{value == 1; list == {}}`
	P1 calls `wait()`, goes into critical section	`S->{value == 0; list == {}}`
	P2 calls `wait()`	`S->{value == -1; list == {P2}}`
	P3 calls `wait()`	`S->{value == -2; list == {P2, P3}}`
T+1:	P1 completes critical section, calls `signal()` and wakes P3	`S->{value == -1; list == {P2}}`
T+2:	P3 completes critical section, calls `signal()` and wakes P2	`S->{value == 0; list == {}}`
T+3:	P2 completes critical section, calls `signal()`	`S->{value == 1; list == {}}`

Liveness

Liveness

Processes may be forced 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, and is an example of liveness failure

Another example of liveness failure is the infinite loop

Liveness refers to a set of properties that a system must satisfy to ensure processes make progress

```
while (true);
```

Deadlock

A liveness failure where two or more processes wait indefinitely for an event that can be caused by only one of the waiting processes

```
P0                P1
wait(S);          wait(Q);
wait(Q);          wait(S);
...               ...
signal(S);        signal(Q);
signal(Q);        signal(S);
```

Let S and Q be two semaphores initialized to 1

Consider if P0 executes `wait(S)` and P1 executes `wait(Q)`

When P0 executes `wait(Q)`, it must wait until P1 executes `signal(Q)`

However, P1 is waiting until P0 execute `signal(S)`

Since these `signal()` operations will never be executed, P0 and P1 are deadlocked

Questions?

Thank You!

[Weihan.Goh {at} Singaporetech.edu.sg]

[https://www.singaporetech.edu.sg/directory/faculty/weihan-goh]

[https://sg.linkedin.com/in/weihan-goh]