CHP#06: Process sync Tools

6.2 What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer

Busy waiting, also known as spinning or busy looping, is a process synchronization technique in which a process or task waits and constantly checks for a condition to be satisfied before proceeding with its execution[1]. In busy waiting, a process executes instructions that test for the entry condition to be true, such as the availability of a lock or resource in the computer system[1].
There are two general approaches to waiting in operating systems: firstly, a process/task can continuously check for the condition to be satisfied while consuming the processor – busy waiting. Secondly, a process can wait without consuming the processor. In such a case, the process/task is alerted or awakened when the condition is satisfied. The latter is known as sleeping, blocked waiting, or sleep waiting[1].
Busy waiting can be inefficient because the looping procedure is a waste of computer resources. In addition, the system is left idle while waiting[1]. A workaround solution for the inefficiency of busy waiting that is implemented in most operating systems is the use of a delay function. Also known as a sleep system call, a delay function places the process involved in busy waiting into an inactive state for a specified amount of time[1]. In this case, resources are not wasted as the process is "asleep"[1].

6.3 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock can be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the program condition required for the first process to make progress[1].
In a multiprocessor system, other processes execute on other processors and thereby modify the program state in order to release the first process from the spinlock[1]. On a multiprocessor system, a thread on another processor may release the lock without you context-switching. Spinlocks can be useful if you don't expect to be waiting long, because it may be faster just to hang around until the other thread unlocks the thing[2].

6.4 Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.

If the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated. Consider the following scenario with two processes, P1 and P2, that share a semaphore S initialized to 1 and a critical section protected by S:

1. P1 executes wait(S) and finds that S is equal to 1. P1 is about to enter the critical section.
2. Before P1 can decrement S and enter the critical section, P2 executes wait(S) and also finds that S is equal to 1.
3. Both P1 and P2 decrement S and enter the critical section simultaneously, violating mutual exclusion.

This scenario is possible because the check of the semaphore value and its decrement are not performed atomically. If these operations were performed atomically, then either P1 or P2 would have successfully entered the critical section while the other would have been blocked until the first process exited the critical section and executed signal(S).

6.5 Illustrate how a binary semaphore can be used to implement mutual exclusion among n processes.

A binary semaphore can be used to implement mutual exclusion among n processes by initializing the semaphore to the value 1 and using it as a lock to protect the critical section. Each process must execute a wait operation on the semaphore before entering the critical section and a signal operation on the semaphore after exiting the critical section.

Here is an example of how this can be done:

```
semaphore mutex = 1; // binary semaphore initialized to 1

// code for process i
while (true) {
    wait(mutex); // acquire the lock
    // critical section
    signal(mutex); // release the lock
    // remainder section
}
```

In this example, only one process can enter the critical section at a time because the binary semaphore `mutex` acts as a lock. When a process executes `wait(mutex)`, it will either acquire the lock and proceed to the critical section if `mutex` is equal to 1, or it will be blocked and wait until another process releases the lock by executing `signal(mutex)`.

This ensures that only one process can enter the critical section at a time, providing mutual exclusion among n processes.

6.6 Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: deposit(amount) and withdraw(amount). These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the withdraw() function, and the wife calls deposit(). Describe how a race condition is possible and what might be done to prevent the race condition from occurring

A race condition is possible in this scenario if the two functions, `withdraw()` and `deposit()`, are not properly synchronized. For example, consider the following sequence of events:

1. The account balance is $100.
2. The husband calls `withdraw(50)` to withdraw $50 from the account.
3. The `withdraw()` function checks the account balance and determines that there are sufficient funds to complete the transaction.
4. Before the `withdraw()` function can update the account balance, the wife calls `deposit(20)` to deposit $20 into the account.
5. The `deposit()` function updates the account balance to $120.
6. The `withdraw()` function then updates the account balance to $50, incorrectly assuming that the account balance is still $100.

In this scenario, the race condition results in an incorrect final account balance of $50 instead of the correct value of $70.

To prevent this race condition from occurring, proper synchronization must be used to ensure that only one function can access and update the account balance at a time. This can be achieved using a variety of synchronization techniques such as locks, semaphores or monitors. For example, a lock can be used to protect access to the account balance:

```
lock balance_lock;

void deposit(int amount) {
    acquire(balance_lock);
    // update account balance
    release(balance_lock);
}

void withdraw(int amount) {
    acquire(balance_lock);
    // check and update account balance
    release(balance_lock);
}
```

In this example, both functions acquire a lock before accessing and updating the account balance and release the lock when they are finished. This ensures that only one function can access and update the account balance at a time, preventing race conditions from occurring.

6.7 The pseudocode of Figure 6.15 illustrates the basic push() and pop() operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:

a. What data have a race condition?

b. How could the race condition be fixed?

```
push(item) {
if (top < SIZE) {
        stack[top] = item;
        top++;
 } else
         ERROR
}

pop() {
if (!is empty()) {
        top--;
        return stack[top];
 } else
        ERROR
}

empty() {
if (top == 0)
        return true;
else
        return false;
 }
```

a. The data that have a race condition in this pseudocode are the shared variables `top` and `stack`. If multiple threads call the `push()` or `pop()` functions concurrently without proper synchronization, race conditions can occur that result in incorrect updates to the `top` and `stack` variables.

b. The race condition can be fixed by using proper synchronization to ensure that only one thread can access and update the shared variables at a time. This can be achieved using a variety of synchronization techniques such as locks, semaphores or monitors. For example, a lock can be used to protect access to the shared variables:

```
lock stack_lock;

void push(int item) {
    acquire(stack_lock);
    if (top < SIZE) {
        stack[top] = item;
        top++;
    } else {
        // ERROR
    }
    release(stack_lock);
}

pop() {
    int item;
    acquire(stack_lock);
    if (!is_empty()) {
        top--;
        item = stack[top];
       return item;
    } else {
        // ERROR
    }
    release(stack_lock);
}

bool is_empty() {
    bool result;
    acquire(stack_lock);
    result = (top == 0);
    release(stack_lock);
    return result;
}
```

In this example, all functions acquire a lock before accessing and updating the shared variables and release the lock when they are finished. This ensures that only one thread can access and update the shared variables at a time, preventing race conditions from occurring.

6.8 Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the bid(amount) function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount) {
        if (amount > highestBid)
                highestBid = amount;
        }
```

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

A race condition is possible in this situation if two or more users attempt to bid at the same time. If their bids are processed concurrently, it's possible that the `bid()` function will be called simultaneously by multiple threads. This could result in the `highestBid` variable being updated incorrectly, as the check `if (amount > highestBid)` and the update `highestBid = amount` are not performed atomically.
One way to prevent this race condition from occurring is to use a synchronization mechanism such as a mutex or a lock to ensure that only one thread can execute the `bid()` function at a time. This would ensure that the check and update of the `highestBid` variable are performed atomically and that the variable is always updated correctly.
Another approach could be to use an atomic compare-and-swap operation to update the `highestBid` variable. This would allow multiple threads to call the `bid()` function concurrently while still ensuring that the variable is updated correctly.

6.10 The compare and swap() instruction can be used to design lock-free data structures such as stacks, queues, and lists. The program example shown in Figure 6.17 presents a possible solution to a lock-free stack using CAS instructions, where the stack is represented as a linked list of Node elements with top representing the top of the stack. Is this implementation free from race conditions?

```
typedef struct node {
 value_t data;
struct node *next;
} Node;
Node *top; // top of stack
void push(value t item) {
Node *old node;
Node *new node;

new node = malloc(sizeof(Node));

new node->data = item;

do {

old node = top;

new node->next = old node;

} while (compare and swap(top,old node,new node) != old node);

}

value_t pop() {

Node *old node;

Node *new node;

do {

old node = top;

if (old node == NULL)

            return NULL;

            new node = old node->next;

} while (compare and swap(top,old node,new node) != old node);

return old node->data;

}
```

6.19 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

• The lock is to be held for a short duration.

• The lock is to be held for a long duration.

• A thread may be put to sleep while holding the lock.

In a system with multiple processing cores, the choice between a spinlock and a mutex lock where waiting processes sleep while waiting for the lock to become available depends on the specific scenario:

- If the lock is to be held for a short duration, a spinlock may be a better choice. This is because the overhead of putting a process to sleep and waking it up again when using a mutex lock can be significant compared to the short time the lock is held. With a spinlock, the waiting process can busy-wait for the lock to become available, avoiding the overhead of sleeping and waking up.

- If the lock is to be held for a long duration, a mutex lock may be a better choice. This is because busy-waiting with a spinlock can waste CPU resources if the lock is held for a long time. With a mutex lock, waiting processes can sleep while waiting for the lock to become available, allowing other processes to use the CPU.
- If a thread may be put to sleep while holding the lock, a mutex lock is generally a better choice. This is because if a thread holding a spinlock is put to sleep, other threads will busy-wait for the lock to become available, wasting CPU resources. With a mutex lock, waiting threads can sleep while waiting for the lock to become available.

#Quiz

Q1. What is a spinlock?

A spinlock is a type of lock that is used to protect shared data in a concurrent system. It is called a "spinlock" because when a thread attempts to acquire the lock and it is already held by another thread, the thread will "spin" or busy-wait until the lock becomes available.

Spinlocks are typically used in low-level system code and in situations where locks are held for very short periods of time. This is because busy-waiting can be more efficient than putting a thread to sleep and waking it up again if the lock is only held for a short time.

However, spinlocks can be inefficient if the lock is held for a long time or if there is contention for the lock, as busy-waiting threads will waste CPU resources. In these situations, other types of locks such as mutexes or semaphores may be more appropriate.

Q2. Requirements to design a solution for critical section.

A critical section is a section of code that accesses shared data and must be executed by only one thread at a time to prevent race conditions and ensure correct behavior. To design a solution for a critical section, you need to use synchronization mechanisms to ensure that only one thread can enter the critical section at a time.

Here are some common synchronization mechanisms that can be used to protect a critical section:

1. Mutex: A mutex is a lock that provides mutual exclusion. Only one thread can acquire the mutex at a time, and other threads must wait until the mutex is released before they can enter the critical section.
2. Semaphore: A semaphore is a synchronization primitive that can be used to control access to a shared resource. A binary semaphore can be used like a mutex to protect a critical section.

3. Monitor: A monitor is a high-level synchronization construct that provides mutual exclusion and condition synchronization. A monitor can be used to protect a critical section by allowing only one thread to execute within the monitor at a time.
4. Atomic operations: Atomic operations are hardware-supported operations that can be used to implement lock-free data structures and algorithms. Atomic operations such as compare-and-swap (CAS) can be used to protect a critical section without the need for locks.

The specific solution you choose will depend on the requirements of your system and the characteristics of your critical section. For example, if your critical section is short and contention for the lock is low, a spinlock may be an appropriate choice. If your critical section is longer or contention is high, a mutex or semaphore may be more appropriate.

Q3. What is bound waiting?

Bound waiting is a condition that must be satisfied to prevent indefinite postponement (also known as starvation) in a concurrent system. It requires that there be a bound or limit 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.

In other words, bound waiting ensures that once a process has requested to enter its critical section, it will not have to wait indefinitely while other processes repeatedly enter and leave their critical sections. Instead, there is a limit on the number of times this can happen before the requesting process is allowed to enter its critical section.

Bound waiting is one of the four conditions (along with mutual exclusion, progress, and no preemption) that must be satisfied to solve the critical section problem and ensure that concurrent processes can access shared resources without conflicts.

Q4. Code for campare_and_swap.

compare_and_swap is an atomic operation that compares the value at a given memory location with a given value and, if they are the same, updates the memory location with a new value. Here is an example implementation of compare_and_swap in C:

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected) {
        *value = new_value;
    }
    return temp;
}
```

This function takes three arguments: a pointer to the memory location to be updated (value), the expected value at that memory location (expected), and the new value to be stored at that memory location (new_value). The function returns the original value at the memory location.

The function first reads the current value at the memory location and stores it in a temporary variable (temp). It then compares the current value with the expected value. If they are equal, it updates the memory location with the new value. Finally, it returns the original value at the memory location.

Note that this implementation is not actually atomic and is provided for illustrative purposes only. In practice, compare_and_swap is typically implemented as a hardware instruction that provides atomicity.