University of Nottingham
UK | CHINA | MALAYSIA

Operating Systems and Concurrency

Lecture 8: Concurrency

University of Nottingham, Ningbo China, 2024

Recap
Last Lecture

- Concurrency issues (e.g. counter++)
- Race conditions,
- Critical sections,
- Critical section problem Solution

Today Class

- Software based:
  - Peterson’s solution
- Hardware based:
  - Disabling Interrupts
  - test_and_set(),
  - compare_and_swap()
- Operating system approach
  - Mutexes

Peterson's Solution
Software Solution

- Is classical software based solution

- Developed in 1981 when simpler memory models and single-core processors dominated.

- Works well for older machines, because of the predictable memory access patterns

- It relies on two shared variables:
  - A boolean array flag[] to indicate whether a process wants to enter the critical section, and
  - a variable turn to indicate whose turn it is to enter the critical section.

- PS restricted to two process (Pi and Pj) that alternate execution between their critical sections and remainder section.

Peterson's Solution
Software Solution (cont.)

- **Shared Variables in Peterson's Solution:**

1. **flag[2]:** An array of boolean values where:
   1. flag[0] indicates whether process P_0 wants to enter the critical section.
   2. flag[1] indicates whether process P_1 wants to enter the critical section.

2. **turn**: A variable that indicates whose turn it is to enter the critical section.
   1. It can either be 0 (for $\mathbf{P_0}$) or 1 (for $\mathbf{P_1}$).
   2. The process whose turn it is will wait until the other process is done with the critical section.

```
do {
    flag[i] = true; // i wants to enter critical section
    turn = j; // allow j to access first
    while (flag[j] && turn == j);
    // whilst j wants to access critical section
    // and its j's turn, apply busy waiting

    // CRITICAL SECTION

    flag[i] = false;

    // REMAINDER SECTION

}while (...);
```

Figure: Peterson's Solution for Process i

```
do {
    flag[j] = true; // j wants to enter critical section
    turn = i; // allow i to access first
    while (flag[i] && turn == i);
    // whilst i wants to access critical section
    // and its i's turn, apply busy waiting

    // CRITICAL SECTION

    flag[j] = false;

    // REMAINDER SECTION

}while (...);
```

Figure: Peterson's Solution for Process j

Peterson's Solution
Software Solution-Example 1

```
PROCESS i
...
flag[i] = true;
turn = j;
...
...
while(flag[j] && turn == j)
...
//CRITICAL SECTION
...
flag[i] = false;
...
...
...
```

```
PROCESS j
...
...
...
flag[j] = true;
turn = i;
...
while(flag[i] && turn == i);
// busy wait
// busy wait
// busy wait
//CRITICAL SECTION
...
flag[j] = false;
```

Peterson’s Solution
Software Solution (Example-2)

```
PROCESS i
...
flag[i] = true;
...
...
turn = j;
while(flag[j] && turn == j)
// busy wait
// busy wait
// busy wait
//CRITICAL SECTION
flag[i] = false;
...
```

```
PROCESS j
...
...
flag[j] = true;
turn = i;
...
...
while(flag[i] && turn == i);
//CRITICAL SECTION
flag[j] = false;
...
...
...
```

Peterson's Solution

Software Solution (Cont.)

```
PROCESS I
...
flag[i] = true;
turn = j;
...
...
while(flag[j] && turn == j)
// busy wait
//CRITICAL SECTION
...
...
flag[i] = false;
...
...
```

```
PROCESS j
...
...
...
flag[j] = true;
...
...
turn = i;
while(flag[i] && turn == i);
// busy wait
// busy wait
// busy wait
//CRITICAL SECTION
flag[j] = false;
```

Peterson’s Solution
Requirements

- Mutual Exclusion: Only one process can enter the critical section at a time.
- Progress: Peterson's Algorithm theoretically ensures progress, real-world issues like scheduling behavior can lead to cases where progress is delayed or where one process might appear to be starved.
- Bounded Waiting: While Peterson’s Solution guarantees bounded waiting in theory, there are practical scenarios where one process could be favored by the system, leading to starvation for the other process.

Peterson’s Solution Benefits

- Peterson’s solution is easy to understand and implement for two processes.
- No Need for Special Hardware: Unlike hardware-based approaches, Peterson’s solution uses simple shared variables and does not rely on atomic instructions or disabling interrupts.

Peterson’s Solution
drawbacks

- Limited to two processes (doesn't scale well):
  - Peterson's solution works only for two processes.
  - Although it can be extended to more processes (e.g., by using more flags and more complex conditions), such extensions are difficult to manage, error-prone, and inefficient.
- Busy waiting leads to inefficient use of CPU resources:
  - Peterson’s solution involves **busy waiting** (also called a spinlock), where a process repeatedly checks if it can enter the critical section.
    - This consumes CPU resources while the process waits.

Peterson’s Solution
drawbacks

- Performance Issues on Modern Architectures
  - Cache Coherence Problems: Peterson's solution assumes that changes made to shared variables (flag[] and turn) are immediately visible to all processors.

    - In modern multi-core systems, memory is often cached locally by each processor. As a result, updates to shared variables might not be visible immediately to other processors due to cache coherence issues, leading to incorrect behavior.

  - Memory Model:
    - Modern processors perform optimizations like instruction reordering and out-of-order execution, which can break the assumptions that Peterson’s solution relies on (i.e., that writes and reads occur in the order specified).

Peterson's Solution
drawbacks

- No fairness or priority mechanism, which may lead to starvation.
    - Starvation: While Peterson's solution ensures mutual exclusion, it does not guarantee fairness.
        - For example, if one process repeatedly sets its flag after completing the critical section, the other process may be forced to wait indefinitely (especially in cases where the system's scheduling behavior favors one process over the other).
    - No Priority:
        - The algorithm does not consider priority between processes. '
            - If two processes have different levels of importance, Peterson's solution cannot ensure that the more important process gets to access the critical section earlier.

Peterson's Solution
drawbacks

- Lack of Deadlock Prevention
  - While it prevents two processes from entering the critical section simultaneously, it doesn't offer mechanisms for avoiding deadlock if both processes are waiting for some external resource in addition to the critical section.
    - E.g. if both **P0** and **P1** are waiting for some **external resource** outside the critical section (for instance, both need a database lock or file access), then neither process may be able to proceed if that resource is not available.

Hardware Approaches
Disabling Interrupts

- When a process enters its critical section, it disables interrupts, ensuring that no context switch, clock interrupt, or other system interrupts can occur while the process is in the critical section.

```
While (true){
/*disable interrupts;*/
/*critical section*/;
/*enable interrupts;*/
/*reminder*/
}
```

Hardware Approaches
Disabling Interrupts: requirements

- Disabling interrupts in a single-processor system can partially fulfill the critical section requirements, but it has limitations.
  - Mutual Exclusion: It guarantees mutual exclusion.
  - Progress: The system as a whole might suffer because other processes, especially those waiting for interrupts (e.g., I/O-bound processes), cannot proceed while interrupts are disabled.
  - Bounded Waiting: This approach does not guarantee bounded waiting. If a process disables interrupts for an extended period, other processes waiting to enter their critical sections or those relying on interrupts could be indefinitely delayed.

Hardware Approaches
Disabling Interrupts

- Benefits:
  - Exclusive access: Once interrupts are disabled, the CPU will not switch to another process, allowing the current process to execute critical operations without interruption.
  - Simplicity: This approach is straightforward and effective for single-processor systems.
- Drawbacks
  - Affects system responsiveness: Disabling interrupts can prevent the system from handling important tasks like I/O or responding to external events in a timely manner.
  - Not scalable: This method is not suitable for multiprocessor systems, as disabling interrupts only affects the local CPU, not other processors.
    - Disabling interrupts on one CPU does not prevent processes on other CPUs from executing or accessing shared resources.

Hardware Approaches
Test and Set Lock (TSL)

- Lock Variable: A shared lock variable (usually a single bit or a memory location) is used to indicate whether the critical section is free (0 for "unlocked") or occupied (1 for "locked").

- Test-and-Set Instruction: The hardware provides a special atomic instruction called test-and-set, which works like this:
  - It checks the current value of the lock.
  - If the lock is free (0), it sets it to locked (1).The operation is atomic, meaning no other process can interfere while this check-and-set operation is in progress.

Atomic Operation

```
// Test and set method
boolean test_and_set(boolean * lock) {
    boolean rv = *lock;
     *lock = true;
    return rv;
}
```

Fig: The definition of the test_and_set() instruction

Hardware Approaches
Test and Set Lock (TSL)

- If a process successfully sets the lock to 1, it can enter the critical section.
- Other processes will be blocked from entering the critical section because the lock is now set.
- Once the process finishes its critical section, it releases the lock by setting it back to 0, allowing other processes to enter.

```
// Test and set method
boolean test_and_set(boolean * lock) {
    boolean rv = *lock;
    *lock = true;
    return rv;
}
```

Fig: The definition of the test_and_set() instruction

- Keep in mind that the initial value of lock is 0/false.

Process P1

```
// Example of using test and set metod
do {
    // while the lock is in use (i.e. true)
    // apply busy waiting
    while (test_and_set(&lock));
    // Lock was false, now true

    // CRITICAL SECTION
    ...
    lock = false;
    ...
     // REMAINDER SECTION
} while (...)
```

Process P2

```
// Example of using test and set metod
do {
    // while the lock is in use (i.e. true)
    // apply busy waiting
    while (test_and_set(&lock));
    // Lock was false, now true

    // CRITICAL SECTION
    ...
    lock = false;
    ...
     // REMAINDER SECTION
} while (...)
```

Hardware Approaches
Test and Set Lock (TSL)(cont.)

- If tes_and_Set() function **is not atomic???**
- Both process can enter critical section at the same time.

```
THREAD 1                        THREAD 2
...                             ...
boolean rv = *lock;             ...
...                             boolean rv = *lock;
*lock = true;                   ...
return rv;                      ...
...                             *lock = true;
...                             return rv;
____                            ____

while (test_and_set(&lock));    while (test_and_set(&lock));
```

Hardware Approaches
Test and Set Lock (TSL) (cont.)

- If test_and_set() is atomic
- Mutual exclusion will be satisfied

```
THREAD 1                          THREAD 2
...                               ...
boolean rv = *lock;               ...
*lock = true;                     ...
return rv;                        ...
...                               boolean rv = *lock;
...                               *lock = true;
...                               return rv;
____                              ____

while (test_and_set(&lock));      while (test_and_set(&lock));
```

Hardware Approach
Test and set lock: requirements

- Mutual Exclusion: The test-and-set lock guarantees that only one process can enter the critical section at a time, fulfilling the mutual exclusion requirement.

- Progress: Progress is satisfied in theory, but in practice, busy-waiting can lead to inefficiencies and delays in a highly contended system. While no process is blocked, progress might be slowed down by the system's scheduling or load.

- Bounded Waiting: It does not inherently guarantee bounded waiting/ fairness. Processes might be stuck in the busy-waiting loop for an indefinite period, which can lead to starvation in certain cases if the lock is continuously held by other processes.

Hardware Approaches
Test and Set Lock (TSL)

- Benefits:
  - Simple and effective: Easy to implement and ensures mutual exclusion.

  - Efficient in small critical sections: Works well when the critical section is short and contention for the lock is low, as the overhead of spinning is minimal.

Hardware Approaches
Test and Set Lock (TSL)

- Drawback:
  - Busy Waiting (Spinlock overhead): Processes busy-waiting for the lock waste CPU resources. This can be inefficient, especially if the critical section is long or there are many threads competing for the lock.
  - Starvation risk: There is no guarantee that all processes will get a chance to access the critical section, especially under heavy contention, leading to potential starvation.
  - Scalability issues: As the number of processes increases, the spinning overhead can become significant, making it less efficient in highly concurrent environments.
  - Deadlock Potential: Incorrect usage of test-and-set locks can lead to deadlocks.
    - E.g., if a process that holds a lock tries to acquire it again, it will block indefinitely, resulting in a deadlock.

Hardware Approaches
Compare_and_Swap() (CAS)

- The CAS function is a hardware-supported atomic instruction

- The compare_and_swap function performs three operations atomically:
  1. Compare: It compares the value at a memory location (lock) with an expected value (expected).
  2. Swap: If the current value in lock matches the expected value, it replaces it with a new value (new_value).
  3. Return: Whether the swap happened or not, it returns the old value at the memory location.

```
int compare_and_swap(int *lock, int expected, int new_value) {
    int temp = *lock;           // Save the current value of lock
    if (*lock == expected)      // Compare lock with the expected value
        *lock = new_value;      // If they match, update lock to new_value
    return temp;                // Return the original value (before swap)
}
```

Fig: The definition of the compare_and_swap function

Arguments:
- lock: Pointer to the shared resource (e.g., a lock) to be checked.
- expected: The value that we expect to find in lock.
- new_value: The value to set in lock if the current value matches expected.

Hardware Approaches
Compare_and_Swap() (CAS)

```c
int compare_and_swap(int *lock, int expected, int new_value) {
    int temp = *lock;        // Save the current value of lock
    if (*lock == expected)   // Compare lock with the expected value
        *lock = new_value;   // If they match, update lock to new_value
    return temp;             // Return the original value (before swap)
}
```

```c
// Shared lock variable initialized to 0 (unlocked)
int lock = 0;
```

```c
// Pseudo-code to use CAS for protecting a critical section
void enter_critical_section() {
    do {
        // Busy-wait until the lock is acquired
        while (compare_and_swap(&lock, 0, 1) != 0) {
            // Spinlock - keep trying to acquire the lock while it is already set
        }

        // CRITICAL SECTION
        // Only one process can execute this part of the code at a time
        // Example: Updating shared data
        print("Process is in the critical section");

        // End of CRITICAL SECTION

        // Release the lock by setting it back to 0 (false/unlocked)
        lock = 0;

    } while (there_is_more_work());  // Repeat if there is more work to do
}
```


```c
// Pseudo-code to use CAS for protecting a critical section
void enter_critical_section() {
    do {
        // Busy-wait until the lock is acquired
        while (compare_and_swap(&lock, 0, 1) != 0) {
            // Spinlock - keep trying to acquire the lock while it is already set
        }

        // CRITICAL SECTION
        // Only one process can execute this part of the code at a time
        // Example: Updating shared data
        print("Process is in the critical section");

        // End of CRITICAL SECTION

        // Release the lock by setting it back to 0 (false/unlocked)
        lock = 0;

    } while (there_is_more_work());  // Repeat if there is more work to do
}
```

Hardware Approach
Compare_and_Swap() (CAS) : requirements

- **Mutual exclusion** is achieved because the compare-and-swap operation is atomic.

- **Progress** can be ensured if implemented with fairness mechanisms, such as backoff strategies or a queue-based CAS approach. However, a basic CAS implementation may also suffer from busy waiting and allow some processes to monopolize access to the critical section, violating progress.

- **Bounded waiting**: CAS does not guarantee bounded waiting, as processes can still be indefinitely delayed by other processes that repeatedly succeed in accessing the critical section.

Hardware Approaches
Compare_and_Swap() (CAS)

- Benefits
  - Multiprocessor Support: CAS is well-supported in modern multiprocessor systems, making it ideal for scalable, high-performance applications.

- Drawback
  - Busy Waiting (Spinlock): Like TST, CAS-based critical sections can cause threads to spin in a loop, wasting CPU resources if the lock is held for a long time.

  - No Fairness: CAS, like TST, does not guarantee fairness or bounded waiting. This can lead to starvation in some scenarios.

OS approaches
Mutex Locks

- **Mutex**(Mutual exclusion) are an approach for mutual exclusion provided by the **operating system** containing a Boolean lock variable to indicate availability
  - The lock variable is set to **true** if the lock is available (process can enter critical section), **false** if not.

- Two atomic functions are used to manipulate the mutex:
  - acquire(): called before entering a critical section, boolean set to false
  - release(): called after exiting the critical section, boolean set to true again

- acquire() and release() must be atomic instructions.

- The **process that acquires** the lock must release the lock**.**

OS approaches
Mutex Locks

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

Figure: Conceptual implementation of aquire()

```
release() {
    available = true;
}
```

Figure: Conceptual implementation of release()

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

Fig: Solution to the CS problem using mutex locks

Mutex Locks Examples (cont.)

```
THREAD 1
...
mutex_lock
sum++;
mutex_unlock
mutex_lock
// busy_wait
sum++
mutex_unlock
mutex_lock
...
```

```
THREAD 2
...
mutex_lock
// busy wait
// busy wait
sum++
mutex_unlock
mutex_lock
// busy wait
sum++
...
```

OS Approach
Mutex: requirements

- Mutual Exclusion: Only one thread can access the critical section at a time.
- Progress:
  - Modern mutex implementations often employ a fair or priority-aware mechanism (depending on the operating system) to ensure that waiting threads are allowed to proceed without indefinite postponement.
- Bounded Waiting: basic mutex implementation does not inherently provide guarantees for bounded waiting.
  - many operating systems and libraries (like pthread_mutex_t) implement mutexes that ensure fairness or employ techniques such as priority inversion avoidance or queuing to prevent starvation.

OS Approach
Mutex Locks

- Benefits
  - Ensures mutual exclusion and prevents race conditions
  - Simple to use with lock/unlock operations
  - Widely supported across platforms and libraries
  - Spinlocks provide an advantage by avoiding costly context switches when the lock is held for a short duration, making them ideal for high-performance, real-time applications.
    - In multiprocessor systems, the spinning thread can be on a different processor, making the overhead of spinning much less problematic.

OS Approach
Mutex Locks

- Drawbacks
  - Can lead to deadlocks if not used correctly.
    - E.g. If thread A locks mutex 1 and waits for mutex 2, and thread B locks mutex 2 and waits for mutex 1, both threads will be stuck, resulting in a deadlock.
  - Risk of priority inversion, where a high-priority task is delayed by a low-priority task holding the mutex.
    - E.g. A high-priority task may be blocked by a low-priority task holding the mutex, leading to longer response times or even system instability.
  - Threads can experience starvation in high-contention scenarios
    - E.g. In a high-contention environment, a thread may never get the chance to acquire the mutex due to other threads constantly acquiring and releasing it, leading to indefinite waiting.

Hardware Approach
Mutex Locks

- Drawbacks
  - Adds locking/unlocking overhead, especially when the critical section is small.
  - Complex with nested locks and requires careful management to avoid deadlocks
  - Mutexes cause threads to block when they cannot acquire the lock(spinlock), which may be undesirable in real-time systems.

Mutex Locks
Linux Examples

//sum is shared variable

```
//includes here
int sum = 0;
pthread_mutex_t lock;

void * calc(void * number_of_increments){
    int i;
    for(i = 0; i < *((int*) number_of_increments);i++)
    {
        pthread_mutex_lock(&lock);
        sum++;
        pthread_mutex_unlock(&lock);
    }
}
int main()
{ int iterations = 50000000;
    pthread_t tid1,tid2;
    pthread_mutex_init(&lock,NULL);
    // no error checking for clarity/brevity
    pthread_create(&tid1, NULL, calc, (void *) &iterations);
    pthread_create(&tid2, NULL, calc, (void *) &iterations);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("the value of sum is: %d\n", sum);
}
```

```
$ gcc -pthread program.c -o program
$ ./program
the value of sum is: 10000000
```

Quiz!

Summary
Take-home Message

- Software based approach: Peterson’s solution (software)
- Hardware based approaches:
  - disabling interrupts
  - atomic instructions: (test_and_set, compare_and_swap)
- OS based approach: Mutexes

Recommended readings

- Modern Operating Systems (Tanenbaum):**Chapter 2(2.3)**

- Operating System Concepts (Silberschatz): **Chapter 6(6.1-5)**

- Operating Systems: Internals and Design Principles (Starlings): **Chapter 5(5.1-2)**