고려대학교
KOREA UNIVERSITY

KU-The Future

Operating Systems (10th Ed., by A. Silberschatz)

Chapter 6 Synchronization Tools

Heonchang Yu
Distributed and Cloud Computing Lab.

Contents

Objectives

- Describe the critical-section problem and illustrate a race condition.
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables.
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical-section problem.
- Evaluate tools that solve the critical-section problem in low-, moderate-, and high-contention scenarios.

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose we want to modify the algorithm to remedy this deficiency (allowed at most BUFFER_SIZE – 1 items in the buffer at the same time).
  - One possibility is to add an integer variable, `count`, initialized to 0.
    - `count` is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

Producer process

```
 while (true)
          /* produce an item and put in next_produced */
     while (count == BUFFER_SIZE)
          ;  /*  do nothing  */
     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
       count++;
}
```

Consumer process

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

Race Condition

- `count++` could be implemented as

  ```
  register1 = count
  register1 = register1 + 1
  count = register1
  ```

- `count--` could be implemented as

  ```
  register2 = count
  register2 = register2 – 1
  count = register2
  ```

- Consider this execution interleaving with "counter = 5" initially:

  T_0: producer execute register1 = **count** {register1 = 5}
  T_1 : producer execute register1 = register1 + 1 {register1 = 6}
  T_2 : consumer execute register2 = **count** {register2 = 5}
  T_3 : consumer execute register2 = register2 - 1 {register2 = 4}
  T_4 : producer execute **count** = register1 {**count** = 6}
  T_5 : consumer execute **count** = register2 {**count** = 4}

Critical-Section Problem

– Each process has a segment of code, called a critical section, in which the process may be accessing – and updating – data that is shared with at least one other process.

– When one process is executing in its critical section, no other process is allowed to execute in its critical section.

– General structure of a typical process
  - Entry section - Each process must request permission to enter its critical section.
  - Exit section

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

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 some processes wish to enter their critical section, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded Waiting** - There exists 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.

Race condition when assigning a pid

- Two processes, P_0 and P_1, are creating child processes using the fork() system call. fork() returns the process identifier of the newly created process to the parent process.
- There is a race condition on the variable kernel variable `next_available_pid` which represents the value of the next available process identifier.
- Unless mutual exclusion is provided, it is possible the same process identifier number could be assigned to two separate processes.

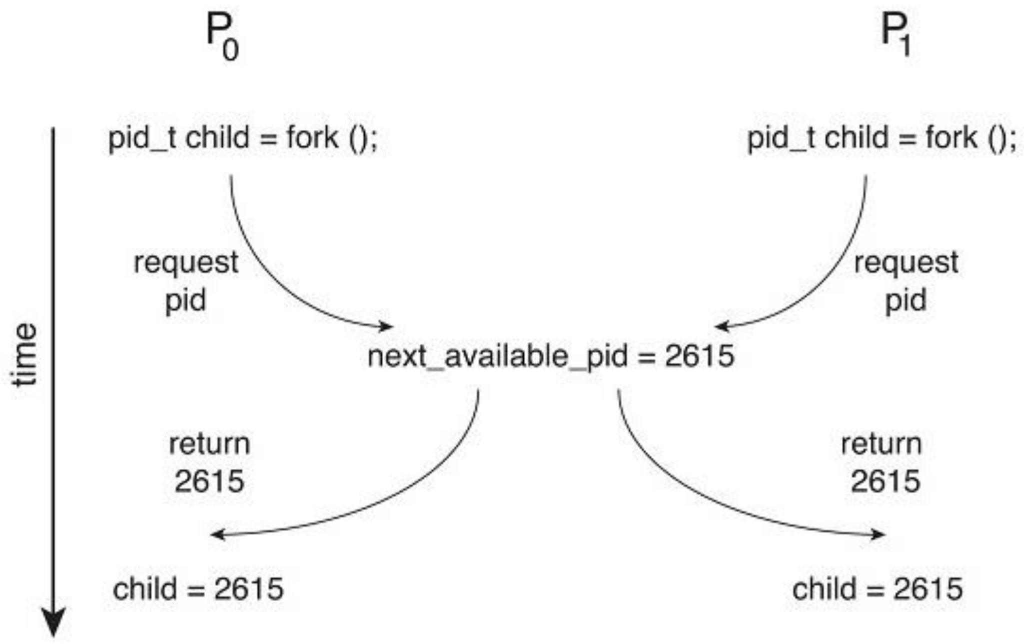


Figure 6.2 Race condition when assigning a pid.

Alternating Algorithm

```
repeat
   while turn $\neq$ i do no-op;
      CRITICAL SECTION
   turn := j;
      REMAINDER SECTION
until false;
```

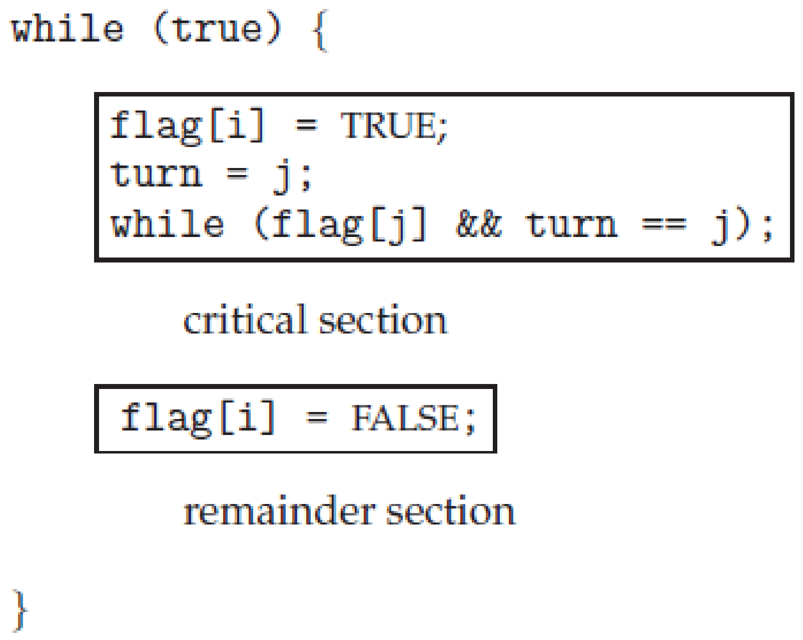
State Testing Algorithm

```
repeat
   flag[i] := true;
   while flag[j] do no-op;

        CRITICAL SECTION

   flag[i] := false;

        REMAINDER SECTION

until false;
```

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two data items:
  - ✓ int turn
  - ✓ Boolean flag[2]
- 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 its critical section. flag[i] = true implies that process P_i is ready to enter its critical section.

Process P_i in Peterson's Solution



```
while (true) {

        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);

            critical section

        flag[i] = FALSE;

            remainder section

}
```

- Mutual exclusion is preserved.
- The progress requirements is satisfied.
- The bounded-waiting requirement is met.

Process P_i in Peterson's Solution

- Consider what happens if the assignments of the first two statements that appear in the entry section of Peterson's solution are reordered.
- It is possible that both threads may be active in their critical sections at the same time, as shown in Figure 6.4.

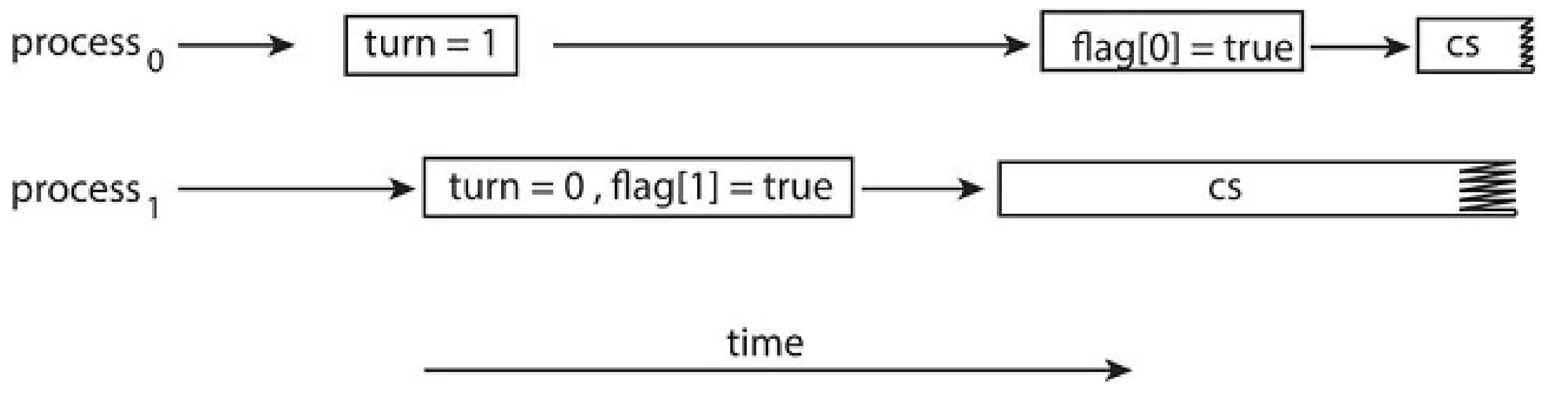


Figure 6.4 The effects of instruction reordering in Peterson's solution.

Hardware Support for Synchronization

- To improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies.
- Example: data that are shared between two threads

```
boolean flag = false;

int x = 0;
```

✓ Thread 1:

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

Thread 2:

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

❖ The expected behavior is that Thread 1 outputs the value 100 for `x`

❖ However, as there are no data dependencies between the variables `flag` and `x`, it is possible that a processor may reorder the instructions for Thread 2 so that `flag` is assigned true before assignment of `x` = 100.

❖ In this situation, it is possible that Thread 1 would output 0 for `x`.

Hardware Support for Synchronization

- Memory model
  - **Strongly ordered**, where a memory modification on one processor is immediately visible to all other processors.
  - **Weakly ordered**, where modifications to memory on one processor may not be immediately visible to other processors.
- Computer architectures provide instructions that can force any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to threads running on other processors.
  - Such instructions are known as **memory barriers** or **memory fences**.
- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Hardware Support for Synchronization

- Reordering of instructions could have resulted in the wrong output
- Use a memory barrier to ensure that we obtain the expected output.

- Thread 1:

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

✓ We guarantee that the value of `flag` is loaded before the value of `x`.

- Thread 2:

```
x = 100;
memory_barrier();
flag = true;
```

✓ We ensure that the assignment to `x` occurs before the assignment to `flag`.

Hardware Support for Synchronization

- Modern computer systems provide special hardware instructions.
    - **Atomic = non-interruptable**
  - ✓ Either to test and modify the content of a word – `test_and_set()`
  - ✓ Or to swap the contents of two words – `compare_and_swap()`
- `test_and_set()` instruction

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

- If two `test_and_set()` instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

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);
```

compare_and_swap( ) Instruction (CAS)

- Mechanism that is based on swapping the content of two words
- If two CAS instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order.
- Definition:

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

Solution using compare_and_swap( )

- A global variable (lock) is initialized to 0;
- When a process exits its critical section, it sets lock back to 0, which allows another process to enter its critical section.
- Solution:

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
       ;  /* do nothing */

       /* critical section */

   lock = 0;

       /* remainder section */

} while (true);
```

swap( ) Instruction

- Definition:

```
void swap (boolean *a, boolean *b)  {
        boolean temp = *a;
        *a = *b;
        *b = temp:
 }
```

Solution using swap

- Shared Boolean variable lock initialized to FALSE.
  - ✓ Each process has a local Boolean variable key.
- Solution:

```
do {
    key = TRUE;
    while ( key == TRUE)
        swap (&lock, &key );
            //    critical section
    lock = FALSE;
        //      remainder section
} while ( TRUE);
```

- Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.

Bounded-waiting mutual exclusion with CAS

- Algorithm using the compare_and_swap() instruction that satisfies all the critical-section requirements
  - ✓ boolean `waiting[n]`   /* initialized to `false`
  - ✓ Int `lock`   /* initialized to 0
- Requirements
  - ✓ mutual-exclusion
    - ❖ Process P_i can enter its critical section only if either `waiting[i] == false` or `key == 0`
  - ✓ progress
    - ❖ A process exiting the critical section either sets `lock` to 0 or sets `waiting[j]` to `false`.
  - ✓ bounded-waiting
    - ❖ It scans the array `waiting` in the cyclic ordering (i+1, i+2, ..., n−1, 0, ..., i−1) and designates the first process in this ordering that is in the entry section (`waiting[j] == true`).

Bounded-waiting mutual exclusion with CAS

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
      key = compare_and_swap(&lock, 0, 1);
  waiting[i] = false;
    /* critical section */
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = 0;
  else
    waiting[j] = false;
    /* remainder section */
}
```

Bounded-waiting mutual exclusion with test_and_set()

- Algorithm using the test_and_set() instruction that satisfies all the critical-section requirements
- Mutual-exclusion
  - ✓ P_i can enter its critical-section only if either waiting[i]==false or key==false.
- Progress
  - ✓ A process exiting the critical section either sets lock to false or sets waiting[j] to false.
- Bounded-waiting
  - ✓ When a process leaves its critical section, it scans the array waiting in the cyclic ordering (i+1, i+2,…, n-1, 0,…, i-1).

Bounded-waiting mutual exclusion with test_and_set()

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
        /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
        /* remainder section */
} while (true);
```

Mutex Locks

- The hardware-based solutions to the C/S problem are complicated as well as generally inaccessible to application programmers.
- OS designers build software tools to solve critical-section problem.
- The simplest of these tools is the **mutex lock**.
  - ✓ Use the mutex lock to protect critical sections and thus prevent race conditions
  - ✓ A process must **acquire** the lock before entering a critical section; it **releases** the lock when it exits the critical section.
  - ✓ Boolean variable (`available`) indicating if lock is available or not
- Calls to **acquire()** and **release()** must be performed atomically.
  - ✓ Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - ✓ This lock is called a **spinlock**
  - ✓ Because the process "spins" while waiting for the lock to become available.

Mutex Locks – acquire() and release()

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

release() {
   available = true;
}


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

Semaphore

- Semaphore *S* – integer variable
- Two standard atomic operations : wait() and signal()
  - ✓ Originally called P() and V()
- Can only be accessed via two indivisible (atomic) operations

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

✓
```
signal (S) {
    S++;
 }
```

Semaphore

- Counting semaphore
  - ✓ integer value can range over an unrestricted domain
- Binary semaphore
  - ✓ Integer value can range only between 0 and 1
  - ✓ Behave similarly to mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Suppose we require that S_2 be executed only after S_1 has completed.
  - ✓ Common semaphore `synch`, initialized to 0

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

Semaphore Implementation with no Busy waiting

- Disadvantage of the semaphore definition – busy waiting
  - ✓ While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. – spinlock
- Disadvantage of the semaphore definition – busy waiting
- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations.
  - ✓ When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However the process can suspend itself.
- Two operations
  - ✓ sleep (suspend) – places a process into a waiting queue associated with the semaphore
  - ✓ wakeup – changes the process from the waiting state to the ready state

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
- Define a semaphore as a "C" struct

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

Semaphore Implementation with no Busy waiting

- Implementation of wait():

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

- Implementation of signal():

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

Semaphore Implementation

- It is critical that semaphores be executed atomically.
- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
  - ✓ In a single-processor environment, we can solve it by inhibiting interrupts during the time wait() and signal() operations are executing.
  - ✓ In a multicore environment, interrupts must be disabled on every processing core. Otherwise, instructions from different processes (running on different cores) may be interleaved in some arbitrary way.
    - ❖ Disabling interrupts on every core can be a difficult task and can seriously diminish performance.

Semaphore Implementation

- We have not completely eliminated busy waiting with this definition of wait() and signal() operations.
  - ✓ We have moved busy waiting from the entry section to the critical sections of application programs.
    - ❖ These sections are short (if properly coded, they should be no more than about ten instructions)
    - ❖ The critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time.

Deadlock and Starvation

- 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

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

- Deadlock (Starvation) – indefinite blocking
  - ✓ Suppose that P0 executes wait(S) and then P1 executes wait(Q).
  - ✓ When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S).

Problems with Semaphores

- Using semaphores incorrectly can result in timing errors that are difficult to detect
  - ✓ An example of such errors in the use of counters in our solution to the producer-consumer problem
- Correct use of semaphore operations:
  - ✓ Interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed

    signal(mutex) .... wait(mutex)

    - ❖ Several processes may be executing in their critical sections simultaneously
  - Replaces signal(mutex) with wait(mutex)

    wait(mutex) ... wait(mutex)

    - ❖ A deadlock will occur
  - Omits the wait(mutex), or signal(mutex), or both