Soongsil Univ.
http://design.ssu.ac.kr

Digital System Design Lab.

Ch. 06 Synchronization Tools

Chanho Lee
Soongsil University

1

Objectives

- Critical-section problem and race condition.
- Hardware solutions to the critical-section problem using
  - ►memory barriers, compare-and-swap operations, and atomic variables.
- Mutex locks, semaphores, monitors, and condition variables : to solve the critical-section problem.
- Evaluate tools that solve the critical-section problem
  - ►in low-, moderate-, and high-contention scenarios.

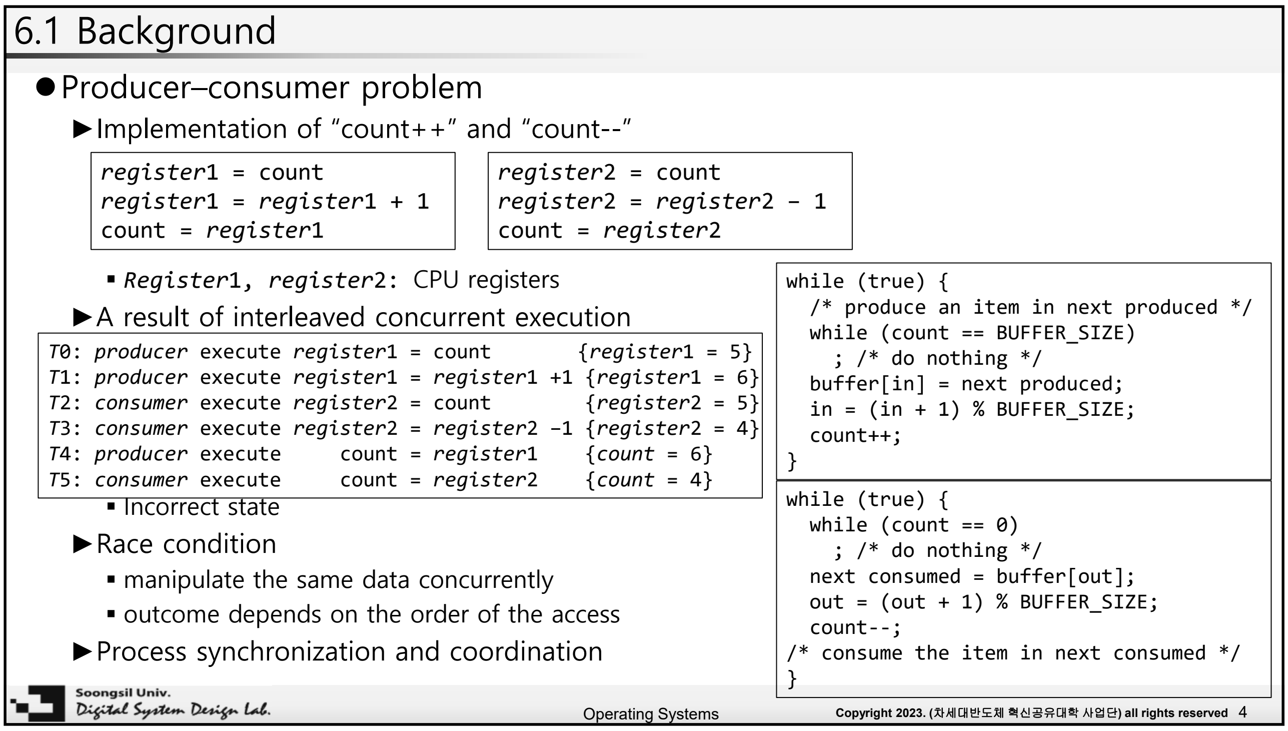
2

Soongsil Univ.
http://design.ssu.ac.kr

Digital System Design Lab.

8주차

3


6.1 Background

- Producer–consumer problem
  - ▶Implementation of "count++" and "count--"

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

```
register2 = count
register2 = register2 - 1
count = register2
```

  - ▪ *Register1, register2*: CPU registers
  - ▶A result of interleaved concurrent execution

```
T0: producer execute register1 = count        {register1 = 5}
T1: producer execute register1 = register1 +1 {register1 = 6}
T2: consumer execute register2 = count        {register2 = 5}
T3: consumer execute register2 = register2 -1 {register2 = 4}
T4: producer execute     count = register1    {count = 6}
T5: consumer execute     count = register2    {count = 4}
```

  - ▪ Incorrect state
  - ▶Race condition
    - ▪ manipulate the same data concurrently
    - ▪ outcome depends on the order of the access
  - ▶Process synchronization and coordination

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

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

4

6.2 The Critical-Section Problem

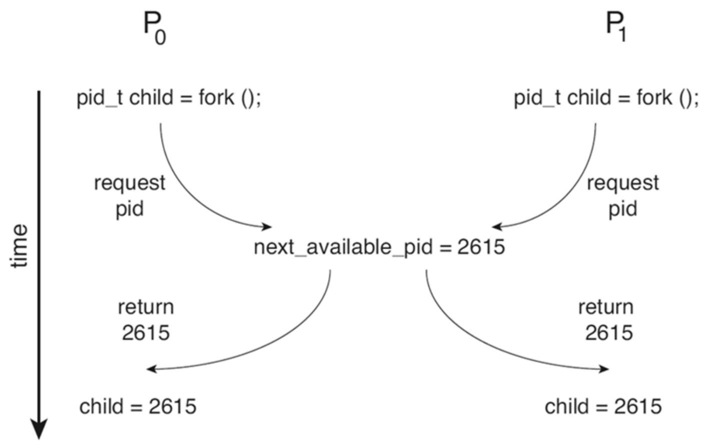
- Critical-section problem
  - ►Critical section: a segment of code in which process may update shared data
    - ▪ Only one process in the critical section
  - ►to design a protocol to synchronize activity so as to cooperatively share data
  - ►general structure of a typical process

```
while (true) {
  entry section
    critical section
  exit section
    remainder section
}
```

  - ►Three requirements for solution to the critical-section problem
    1. Mutual exclusion: only one process in the critical section
    2. Progress: processes in entry section can participate in deciding to enter its critical section next, and this selection cannot be postponed indefinitely
    3. Bounded waiting: after a request and before a grant

6.2 The Critical-Section Problem

- Ex. 1: a kernel data structure maintaining a list of all open files
  - ►Simultaneous opening files by two processes
  - ►Separate updates could result in a race condition
- Ex. 2: Simultaneous creating child processes



6.2 The Critical-Section Problem

- Ex. 3: kernel data structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling
- Solution in a single-core environment
  - ►In order execution without preemption: to prevent interrupts while a shared variable was being modified
- Critical section handling in OS
  - ►Preemptive kernels: allows a process preempted while running in kernel mode
    - difficult to design for SMP architectures
    - more responsive
    - more suitable for real-time programming
  - ►Nonpreemptive kernel: a kernel-mode process will run until
    - it exits kernel mode, blocks, or voluntarily yields control of the CPU
    - free from race conditions

6.3 Peterson's Solution

- a classic software-based solution
  - ►May not work for the modern processor; provides a good algorithmic description
  - ►restricted to two processes that alternate execution between critical and remainder sections
- Structure of process Pi

```
int turn; //Who enters critical section
boolean flag[2];
while (true) {
  flag[i] = true; //Pi is ready to enter its critical section
  turn = j; //Pj is allowed to execute in its critical section
  while (flag[j] && turn == j) //True if arriving later
    ;                           // stay in the while
    /* critical section */
  flag[i] = false;
    /*remainder section */
}
```

```
int turn;
boolean flag[2];
while (true) {
  flag[j] = true;
  turn = i;
  while (flag[i] && turn == i)
    ;
    /* critical section */
  flag[j] = false;
    /*remainder section */
}
```

6.3 Peterson's Solution

- Structure of process Pi

```
int turn; //Who enters critical section
boolean flag[2];
while (true) {
  flag[i] = true; //Pi is ready to enter its critical section
  turn = j; //Pj is allowed to execute in its critical section
  while (flag[j] && turn == zj) //True if arriving later
    ;                           // stay in the while
    /* critical section */
  flag[i] = false;
    /*remainder section */
}
```

```
int turn;
boolean flag[2];
while (true) {
  flag[j] = true;
  turn = i;
  while (flag[i] && turn == i)
    ;
    /* critical section */
  flag[j] = false;
    /*remainder section */
}
```

- ▶Requirements
  - Mutual exclusion: turn can be either 0 or 1 but cannot be both → satisfied
  - Progress: single entry – simple.
    - Both – Pi arrives first and stays in the while → turn is set by Pj → Pi entry
  - Bounded-waiting: single entry – simple
    - Both – Pi waiting → `flag[j] = false` when Pj exits → Pi entry (waiting for at most one entry)

6.3 Peterson's Solution

- Reordering of instructions by processors and/or compilers
  - ▶Single-threaded: working correctly
  - ▶Multithreaded application with shared data
    - Ex.

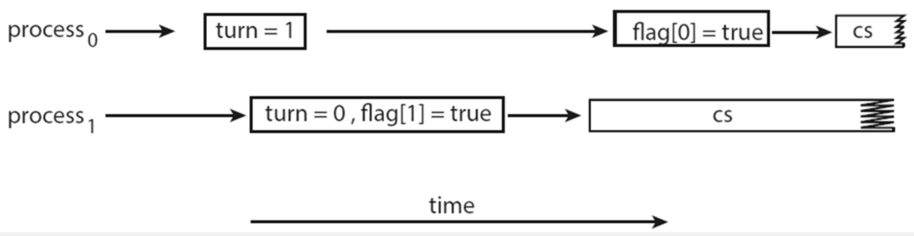
```
//Data
boolean flag = false;
int x = 0;
```

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

```
//Thread 2
x = 100;
flag = true;
// no data dependency
```

- Correct answer: 100
- Reordering in thread 2: 0 or 100
- Reordering in thread 1: 0 or 100

- Peterson's solution



6.4 Hardware Support for Synchronization

- Software-based solutions are not guaranteed to work on modern computer architectures
- Memory Barriers
  - ▶memory model
    - Strongly ordered: a memory modification on one processor is immediately visible to all other processors.
    - Weakly ordered: modifications to memory on one processor may not be immediately visible to other processors.
  - ▶Memory barriers (or memory fences): instructions that can force any changes in memory to be propagated
    - ensures that all loads and stores are completed before any subsequent load or store operations
    - Ex.

```
//Data
boolean flag = false;
int x = 0;
```

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

the value of flag is loaded before the value of x

```
//Thread 2
x = 100;
memory barrier();
flag = true;
```

the assignment to x occurs before the assignment to flag

to avoid the reordering of operations

6.4 Hardware Support for Synchronization

- Hardware Instructions
  - ▶to test and modify the content of a word or to swap the contents of two words atomically—that is, as one uninterruptible unit
  - ▶Conceptual instruction: `test and set()` and `compare and swap()`
    - Simultaneous execution on different core: sequential execution in arbitrary order

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


do {
  while (test_and_set(&lock))
    ; /* do nothing */
    /* critical section */
  lock = false;
    /* remainder section */
} while (true);
```

6.4 Hardware Support for Synchronization

- Hardware Instructions (cont.)
  - ▶CAS ex.
    - 2nd process cannot access to critical section

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

```
while (true) {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ; /* do nothing */
    /* critical section */
  lock = 0;
    /* remainder section */
}
//Satisfying mutual-exclusion
//Not satisfying bounded-waiting
```

```
boolean waiting[n]; // initialized to false
int lock; // initialized to 0
```

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

6.4 Hardware Support for Synchronization

- Atomic Variables
  - ▶provides atomic operations on basic data types such as integers and booleans
  - ▶Ex. increment(&sequence); → atomic integer

```
void increment(atomic_int *v)
{
  int temp;
  do {
  temp = *v;
  }
  while (temp != compare_and_swap(v, temp, temp+1));
}
```

  - ▶atomic variables do not entirely solve race conditions in all circumstances
    - Ex. bounded-buffer problem (producer-consumer)
      - Empty buffer, two consumers waiting
      - Producer provides one item → both consumers could exit their while loops

6.5 Mutex Locks

- Hardware-based solutions
  - complicated as well as generally inaccessible to application programmers
- Mutex(mutual exclusion) lock
  - Simplest higher-level software tools
  - protects critical sections and prevents race conditions

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

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

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

  - Calls to either `acquire()` or `release()` must be performed atomically
    - implemented using the CAS operation
  - busy waiting: waiting processes loop continuously in the call to `acquire()`
  - Spinlock: the processes "spins" while waiting.
    - No context switch → preferable for multicore systems (widely used)

6.6 Semaphores

- A more robust tool for synchronization in more sophisticated ways (Dijkstra)
- Semaphore S
  - integer variable: accessed only through two atomic operations: `wait()` and `signal()`
  - wait(): termed P (from proberen, "to test")
  - signal(): called V (from verhogen, "to increment")

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

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

  - Atomic execution on all modifications of the semaphore
    - `wait()`: S<=0 and S-- without interruption

6.6 Semaphores

- Semaphore Usage
  - ▶Counting semaphore: over an unrestricted domain
    - Ex. to control access to a given resource consisting of a finite number (→initialized) of instances
      - `wait()`: decrementing the count until 0 → resource blocked
      - `signal()`: incrementing the count
    - Ex. two concurrently running processes: P1 with a statement S1 and P2 with S2: S2 after S1
      - semaphore synch=0

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

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

  - ▶Binary semaphore: 0 and 1
    - Similar to mutex lock

6.6 Semaphores

- Semaphore Implementation
  - ▶Modified wait() and signal(): preventing busy waiting
  - ▶Waiting process is suspended in the waiting queue associated with the semaphore: waiting state
    - Processes are added to and removed from S->list
  - ▶Restarted by `signal()` operation of another process
    - waiting → ready
  - ▶S->value may be negative
    - number of processes waiting on that semaphore
  - ▶Atomic execution of wait() and signal()
    - Disable interrupts on every core or
    - Provide compare_and_swap() or spinlocks

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

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

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

6.7 Monitors

- Synchronization problem due to programming errors
  - ►Simultaneous entry to critical section
  - ►Permanent blocking

```
signal(mutex);
...
critical section
...
wait(mutex);
```

```
wait(mutex);
...
critical section
...
wait(mutex);
```

```
wait(mutex);
...
critical section
...
```

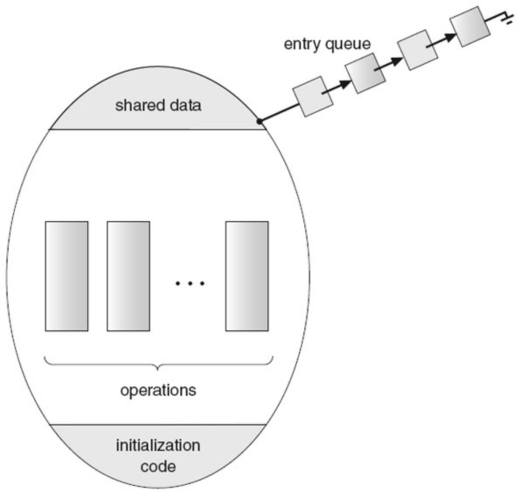
```
...
critical section
...
signal(mutex);
```

  - ►A solution: to incorporate simple synchronization tools as high-level language constructs
- Monitor Usage
  - ►`monitor` type: an abstract data type (ADT)
    - includes a set of programmer-defined operations
    - provided with mutual exclusion within the monitor
    - declares the variables: define the state of an instance of that type
      + bodies of functions: operate on those variables
    - cannot be used directly by the various processes
      – Variables and functions are local

6.7 Monitors

- Monitor usage (cont.)
  - ►Only one process at a time is active within a monitor
  - ►not sufficiently powerful for modeling some synchronization schemes

```
monitor monitor name
{
  /* shared variable declarations */
  function P1 ( . . . ) {
  . . .
  }
  function P2 ( . . . ) {
  . . .
  }
  .
  .
  .
  function Pn ( . . . ) {
  . . .
  }
  initialization code ( . . . ) {
  . . .
  }
}
```

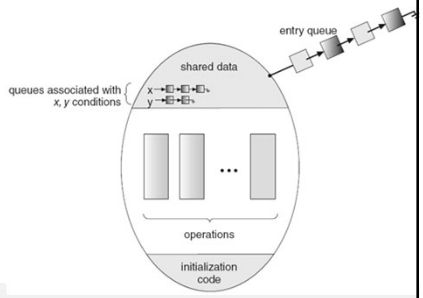


6.7 Monitors

- Monitor usage (cont.)
  - ► condition construct for tailor-made synchronization

```
condition x, y;
x.wait(); //process is suspended
x.signal(); //release a suspended process. No effect without any waiting process
```

  - ▪ cf. signal() in semaphore: always affects the state of the semaphore
  - ►Q: `suspended`, P: `x.signal():` Q is resumed → 2 possibilities
    1. Signal and wait. P either waits until Q leaves the monitor or waits for another condition
    2. Signal and continue. Q either waits until P leaves the monitor or waits for another condition



6.7 Monitors

- Implementing a Monitor Using Semaphores
  - ►Implementation of function F (signal-and-wait scheme)

```
wait(mutex); //binary semaphore mutex (initialized to 1) for mutual exclusion
  ...
  body of F
  ...
if (next_count > 0) //integer next_count : to count the number of processes suspended on next
  signal(next); //binary semaphore next (initialized to 0)
else
  signal(mutex);
```

```
//x.wait()
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
X_count--;
```

```
//x.signal
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next); //signal and wait
  next_count--;
}
```

Binary semaphore

6.7 Monitors

- Resuming Processes within a Monitor
  - Simple resuming scheduling: FCFS, or
  - Conditional-wait construct: `x.wait(c);` //c: priority number
    - With `x.signal()`, process with the smallest priority number is resumed next

Summary

- Race condition
  - Race conditions can result in corrupted values of shared data.
- Critical section: shared data and a possible race condition
- A solution to the critical-section problem:
  - (1) mutual exclusion, (2) progress, and (3) bounded waiting.
- Software solutions: Peterson's solution,
- Hardware support for the critical-section problem
  - memory barriers, hardware instructions(CAS and atomic variables)
- Mutex lock: mutual exclusion, acquire a lock
- Semaphores: mutual exclusion, integer value
- Monitor: high-level form of process synchronization. condition variables

Exercises, problems and projects

- Exercises
  - ►6.2, 6.3, 6.6
- Problems
  - ►6.7, 6.9, 6.12, 19

6.7 The pseudocode below 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
}
```

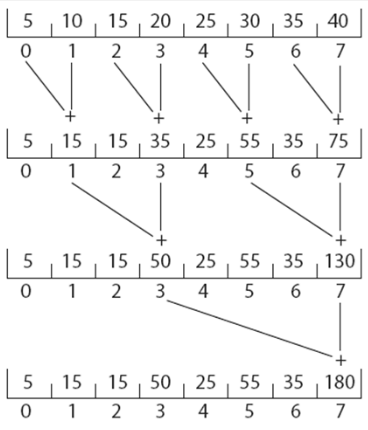
```
is empty() {
  if (top == 0)
    return true;
  else
    return false;
}
```

Exercises, problems and projects

6.9 The following program example can be used to sum the array values of size *N* elements in parallel on a system containing *N* computing cores (there is a separate processor for each array element):

```
for j = 1 to log 2(N) {
  for k = 1 to N {
    if ((k + 1) % pow(2,j) == 0) {
      values[k] += values[k - pow(2,(j-1))]
    }
  }
}
```

This has the effect of summing the elements in the array as a series of partial sums, as shown in the right side. After the code has executed, the sum of all elements in the array is stored in the last array location. Are there any race conditions in the above code example? If so, identify where they occur and illustrate with an example. If not, demonstrate why this algorithm is free from race conditions.



Exercises, problems and projects

6.12 Some semaphore implementations provide a function getValue() that returns the current value of a semaphore. This function may, for instance, be invoked prior to calling wait() so that a process will only call wait() if the value of the semaphore is > 0, thereby preventing blocking while waiting for the semaphore. For example:

```
if (getValue(&sem) > 0)
  wait(&sem);
```

Many developers argue against such a function and discourage its use. Describe a potential problem that could occur when using the function getValue() in this scenario.

6.19 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a 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.

Exercises, problems and projects

- Programming Problems

6.33 Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and will return them once finished. As an example, many commercial software packages provide a given number of licenses, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such a request will be granted only when an existing license holder terminates the application and a license is returned.
The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

When a process wishes to obtain a number of resources, it invokes the decrease count() function:

```
/* decrease available resources by count resources return 0 */
/* if sufficient resources available, otherwise return -1 */
int decrease count(int count) {
  if (available_resources < count) return -1;
  else {
    available_resources -= count;
    return 0;
  }
}
```

Exercises, problems and projects

- Programming Problems (cont.)

When a process wants to return a number of resources, it calls the increase_count() function:

```
/* increase available resources by count */
int increase_count(int count) {
  available_resources += count;
  return 0;
}
```

The preceding program segment produces a race condition. Do the following:

a. Identify the data involved in the race condition.
b. Identify the location (or locations) in the code where the race condition occurs.
c. Using a semaphore or mutex lock, fix the race condition. It is permissible to modify the decrease count() function so that the calling process is blocked until sufficient resources are available.