



Synchronization (I)

Operating Systems

Yu-Ting Wu

1

Operating Systems 2022

Outline

- Background
- Critical section
- Synchronization hardware
- Semaphores

2

2

Operating Systems 2022

Background

3

3

Operating Systems 2022

Background

- **Concurrent access** to **shared data** may result in **data inconsistency**
- Maintaining data consistency requires mechanism to ensure the **orderly execution** of cooperating processes

4

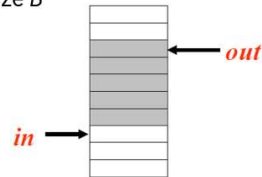
4

Consumer & Producer Problem

- Determine whether buffer is empty or full
 - Use *in*, *out* position

- Buffer as a circular array with size *B*

- Next free: *in*
- First available: *out*
- Empty: *in* = *out*
- Full: $(in + 1) \% B = out$



- The solution allows at most $(B - 1)$ item in the buffer
 - Otherwise, cannot tell the buffer is empty or full

5

Consumer & Producer Problem (cont.)

- Determine whether buffer is empty or full
 - Use *count* value

```

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

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

```

6

Concurrent Operations on Counter

- The statement "counter++" may be implemented in machine language as

```

move ax, counter
add ax, 1
move counter, ax

```

- The statement "counter--" may be implemented as

```

move bx, counter
sub bx, 1
move counter, bx

```

7

Instruction Interleaving

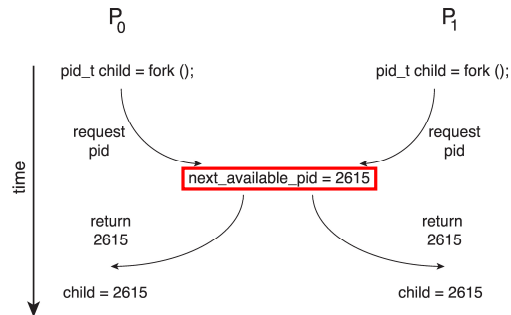
- Assume counter is initially 5. One interleaving of statement is

producer: move ax, counter	→ ax = 5
producer: add ax, 1	→ ax = 6
context switch	
consumer: move bx, counter	→ bx = 5
consumer: sub bx, 1	→ bx = 4
context switch	
producer: move counter, ax	→ counter = 6
context switch	
consumer: move counter, bx	→ counter = 4

8

Another Example

- An example in the kernel



9

Race Condition

- The situation where several processes access and manipulate **shared** data concurrently.
- The final value of the shared data depends upon which process finishes last**
- To prevent race condition, concurrent processes must be **synchronized**
 - On a single-processor machine, we could disable interrupt or use non-preemptive CPU scheduling
 - But how about on **multi-processor** machines and **preemptive** scheduling?
- We need a mechanism to solve the synchronization issue, commonly described as **critical section problem**

10

10

Critical Section

11

11

The Critical-Section Problem

- Purpose**
 - A **protocol** for processes to cooperate
- Problem description**
 - N processes are competing to use some **shared** data
 - Each process has a **code segment**, called **critical section**, in which the shared data is accessed
 - Ensure that when one process is executing in its critical section, **no other process is allowed** to execute in its critical section
 - **mutually exclusive !**

12

12

The Critical-Section Problem (cont.)

• General code section structure

- Only one process can be in a critical section

```
do {
    entry section      → get entry permission
    critical section   → modified shared data
    exit section       → release entry permission
    remainder section
} while (1);
```

13

Critical-Section Requirements

• Mutual exclusion

- If a process P is executing in its critical section (CS), no other processes can be executing in their CS

• Progress

- If no process is executing in its CS and there exist some processes that wish to enter their CS, these processes cannot be postponed indefinitely

• Bounded Waiting

- A **bound** must exist on the number of times that other processes are allowed to enter their CS after a process has made a request to enter its CS
- How to design **entry** and **exit** section to satisfy the above requirement?

14

13

14

CS Solutions and Synchronization Tools

- Software solution
- Synchronization hardware
- Semaphore
- Monitor

15

Algorithm for Two Processes

- Only 2 processes P_0 and P_1
- Shared variables
 - int $turn$; // initially $turn = 0$
 - $turn == i \rightarrow P_i$ can enter its critical section

```
/* Process 0 */
do {
    while (turn != 0); — entry section — while (turn != 1);
    critical section
    turn = 1; — exit section — turn = 0;
    remainder section
} while (1);

/* Process 1 */
do {
    while (turn != 1); — entry section — while (turn != 0);
    critical section
    turn = 0; — exit section — turn = 1;
    remainder section
} while (1);
```

mutual exclusion? **Y** progress? **N** bounded-wait? **Y**

16

15

16

Peterson's Solution for Two Processes

• Shared variables

- `int turn;` // initially `turn = 0`
- `turn == i` → P_i can enter its critical section
- `boolean flag[2];` // initially `flag[0] = flag[1] = false`
- `flag[i] == true` → P_i is ready to enter its critical section

```
/* Process i */
do {
    flag[i] = true;
    turn = i;
    while (flag[j] && turn == j);
    critical section
    flag[i] = false;
    remainder section
} while (1);
```

— entry section

— exit section

17

17

Proof of Peterson's Solution

• Mutual exclusion

- If P_0 in CS → `flag[1] == false` || `turn == 0`
- If P_1 in CS → `flag[0] == false` || `turn == 1`
- Assume both processes in CS → `flag[0] == flag[1] == true`
 - `turn == 0` for P_0 to enter, `turn == 1` for P_1 to enter
 - `turn` will be either 0 or 1, so P_0, P_1 cannot in CS at the same time

```
/* Process 0 */
do {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
    critical section
    flag[0] = false;
    remainder section
} while (1);

/* Process 1 */
do {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
    critical section
    flag[1] = false;
    remainder section
} while (1);
```

18

18

Proof of Peterson's Solution

• Progress (e.g., P_0 wishes to enter its CS)

- (1) If P_1 is not ready → `flag[1] = false` → P_0 can enter
- (2) If both are ready → `flag[0] == flag[1] == true`
 - If `turn == 0` then P_0 enters, otherwise P_1 enters
- Either cases, some waiting process can enter CS

```
/* Process 0 */
do {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
    critical section
    flag[0] = false;
    remainder section
} while (1);

/* Process 1 */
do {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
    critical section
    flag[1] = false;
    remainder section
} while (1);
```

(2)

(1)

19

19

Proof of Peterson's Solution

• Bounded waiting (e.g., P_0 wishes to enter its CS)

- (1) Once P_1 exits CS → `flag[1] == false` → P_0 can enter
- (2) If P_1 exits CS and reset `flag[1] = true`
 - `turn == 0` (overwrite P_0 setting) → P_0 can enter
- P_0 won't wait infinitely

```
/* Process 0 */
do {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
    critical section
    flag[0] = false;
    remainder section
} while (1);

/* Process 1 */
do {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
    critical section
    flag[1] = false;
    remainder section
} while (1);
```

(2)

(1)

20

20

Peterson's Solution and Modern Architecture

- Peterson's solution is **not guaranteed** to work on modern architectures
 - To improve performance, processors and/or compilers may **reorder operations** that have no **dependencies**
- For **single-threaded process** this is **OK** as the result will always be the same
- For **multi-threaded process** the reordering may produce inconsistent or unexpected results

21

21

Peterson's Solution and Modern Architecture (cont.)

- Example:
 - Two threads share the data:


```
bool flag = true;
int x = 0;
```
 - Thread1 performs


```
while (!flag);
print x;
```
 - Thread2 performs


```
x = 100;
flag = true;
```
 - Expected output will be 100

22

22

Peterson's Solution and Modern Architecture (cont.)

- Example (cont.):
 - Because the variables **flag** and **x** are independent of each other, the instructions:


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

 for Thread2 may be reordered
 - If this occurs, the output may be 0!

23

23

Peterson's Solution and Modern Architecture (cont.)

```

/* Process 0 */
do {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
    critical section
    flag[0] = false;
    remainder section
} while (1);

/* Process 1 */
do {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
    critical section
    flag[1] = false;
    remainder section
} while (1);

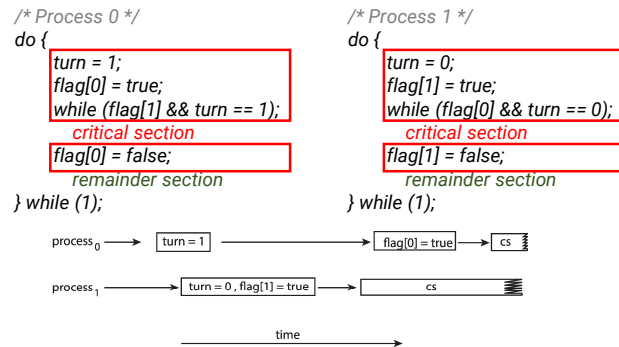
```

The variables **flag[]** and **turn** are independent, so they might be reordered

24

24

Peterson's Solution and Modern Architecture (cont.)



Both processes will enter their CS !
We can use **Memory Barrier** to ensure the correctness

25

Memory Barrier

- When a memory barrier instruction is performed, the system ensures that **all loads and stores are completed before any subsequent loads or stores operations are performed**
- Recall previous example:

```

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

/* Thread 2 */
x = 100; store
flag = true;

```

26

Memory Barrier (cont.)

- When a memory barrier instruction is performed, the system ensures that **all loads and stores are completed before any subsequent loads or stores operations are performed**
- Modification:

```

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

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

```

For Thread 1, we are guaranteed that the value of *flag* is loaded before the value of *x*

27

Memory Barrier (cont.)

- 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**
- Modification:

```

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

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

```

For Thread 2, we are guaranteed that the assignment to *x* occurs before the assignment to *flag*

28

Producer & Consumer Problem

```

/* Producer process 0 */
while (true) {
    entry section
    nextItem = getItem();
    while (counter == BUFFER_SIZE);
    buffer[in] = nextItem;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
    computing();
    exit section
}

/* Consumer process 0 */
while (true) {
    entry section
    while (counter == 0);
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    computing();
    exit section
}

```

Incorrect. Deadlock if consumer enters the CS first

29

29

Producer & Consumer Problem (cont.)

```

/* Producer process 0 */
while (true) {
    nextItem = getItem();
    while (counter == BUFFER_SIZE);
    buffer[in] = nextItem;
    in = (in + 1) % BUFFER_SIZE;
    entry section
    counter++;
    computing();
    exit section
}

/* Consumer process 0 */
while (true) {
    while (counter == 0);
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    entry section
    counter--;
    computing();
    exit section
}

```

Correct but **poor performance**

30

30

Producer & Consumer Problem (cont.)

```

/* Producer process 0 */
while (true) {
    nextItem = getItem();
    while (counter == BUFFER_SIZE);
    buffer[in] = nextItem;
    in = (in + 1) % BUFFER_SIZE;
    entry section
    counter++;
    exit section
    computing();
}

/* Consumer process 0 */
while (true) {
    while (counter == 0);
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    entry section
    counter--;
    exit section
    computing();
}

```

Correct and **maximize concurrent performance**

31

31

Bakery Algorithm (n processes)

- Before entering its CS, each process receives a **number (#)**
- **Holder of the smallest # enters CS**
- The numbering scheme always generates # in **non-decreasing order**; i.e., 1, 2, 3, 3, 4, 5, 5, 5 ...
- If processes P_i and P_j receive the same #, if $i < j$, then P_i is served first
- Notation:
 - $(a, b) < (c, d)$ if
 - $a < c$ or
 - $a == c$ && $b < d$

32

32

Bakery Algorithm (n processes) (cont.)

```
// Process i:
do {
    get ticket → choosing[i] = true;
                num[i] = max (num[0], num[1], ...num[n-1]) + 1;
                choosing[i] = false;
    FCFS → for (j = 0; j < n; j++) {
                while (choosing[j]);
                while ((num[j] != 0) && ((num[j], j) < (num[i], i)));
            }
    release ticket → critical section
                    num[i] = 0;
                    remainder section
}

Bounded waiting because processes enter CS on a first come, first served basis
```

33

Bakery Algorithm (n processes) (cont.)

- Why cannot compare when num is being modified?
- Without locking
 - Let 5 be the current maximum number
 - If P_1 and P_4 take number together, but P_4 finishes before P_1
 - $P_1 = 0, P_4 = 6 \rightarrow P_4$ will enter the CS
 - After P_1 takes the number
 - $P_1 = P_4 = 6 \rightarrow P_1$ will enter the CS as well !
- With locking
 - P_4 will have to wait until P_1 finish taking the number
 - Both P_1 & P_4 will have the new number "6" before comparison

34

Pthread Lock/Mutex Routines

- To use mutex, it must be declared as of type `pthread_mutex_t` and initialized with `pthread_mutex_init()`
- A mutex is destroyed with `pthread_mutex_destroy()`
- A critical section can then be protected using `pthread_mutex_lock()` and `pthread_mutex_unlock()`

```
#include "pthread.h"
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
pthread_mutex_lock(&mutex);      ← enter critical section
critical section
pthread_mutex_unlock(&mutex);    ← exit critical section
pthread_mutex_destroy(&mutex);
```

35

Condition Variables

- **Condition variables (CV)** represent some condition that a **thread** can
 - **Wait on**, until the condition occurs; or
 - **Notify** other waiting threads that the condition has occurred
- Three operations on condition variables
 - **wait()** – block until another thread calls **signal()** or **broadcast()** on the CV


```
pthread_cond_wait(&theCV, &someLock)
```
 - **signal()** – wake up one thread waiting on the CV


```
pthread_cond_signal(&theCV)
```
 - **broadcast()** – wake up all threads waiting on the CV


```
pthread_cond_broadcast(&theCV)
```

36

Condition Variables (cont.)

• Example

- A thread is designed to take action when $x == 0$
- Another thread is responsible for decrementing the counter

```
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

```
action() {
    pthread_mutex_lock(&mutex);
    if (x != 0)
        pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);
    take_action();
}

counter() {
    pthread_mutex_lock(&mutex);
    x--;
    if (x == 0)
        pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
```

37

37

Using Condition Variables

```
action() {
    pthread_mutex_lock(&mutex);
    if (x != 0)
        pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);
    take_action();
}

counter() {
    pthread_mutex_lock(&mutex);
    x--;
    if (x == 0)
        pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
```

Lock mutex

38

38

Using Condition Variables

```
action() {
    pthread_mutex_lock(&mutex);
    if (x != 0)
        pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);
    take_action();
}

counter() {
    pthread_mutex_lock(&mutex);
    x--;
    if (x == 0)
        pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
```

Lock mutex

Wait()

- Put the thread into **sleep** and **releases the lock**

Lock mutex

39

39

Using Condition Variables

```
action() {
    pthread_mutex_lock(&mutex);
    if (x != 0)
        pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);
    take_action();
}

counter() {
    pthread_mutex_lock(&mutex);
    x--;
    if (x == 0)
        pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
```

Lock mutex

Wait()

- Put the thread into **sleep** and **releases the lock**
- **Waked up**, but the thread is locked

Lock mutex

Signal()

40

40

Operating Systems 2022

Using Condition Variables

```

action() {
    pthread_mutex_lock(&mutex);
    if (x != 0)
        pthread_cond_wait(cond, mutex);
    pthread_mutex_unlock(&mutex);
    take_action();
}

counter() {
    pthread_mutex_lock(&mutex);
    x--;
    if (x == 0)
        pthread_cond_signal(cond);
    pthread_mutex_unlock(&mutex);
}

```

Lock mutex
Wait()

- Put the thread into **sleep** and **releases the lock**
- Waked up**, but the thread is locked
- Re-acquire lock** and resume execution

Lock mutex
Signal()
 Release the lock

41

Operating Systems 2022

Using Condition Variables

```

action() {
    pthread_mutex_lock(&mutex);
    if (x != 0)
        pthread_cond_wait(cond, mutex);
    pthread_mutex_unlock(&mutex);
    take_action();
}

counter() {
    pthread_mutex_lock(&mutex);
    x--;
    if (x == 0)
        pthread_cond_signal(cond);
    pthread_mutex_unlock(&mutex);
}

```

Lock mutex
Wait()

- Put the thread into **sleep** and **releases the lock**
- Waked up**, but the thread is locked
- Re-acquire lock** and resume execution

 Release the lock

Lock mutex
Signal()
 Release the lock

42

41

42

Operating Systems 2022

ThreadPool Implementation

- Task structure**

```

typedef struct {
    void (*function)(void *);
    void *argument;
} threadpool_task_t;

```
- ThreadPool structure**

```

struct threadpool_t {
    pthread_mutex_t lock;
    pthread_cond_t notify;
    pthread_t *threads;
    threadpool_task_t *queue;
    int thread_count;
    int queue_size;
    int head;
    int tail;
    int count;
    int shutdown;
    int started;
};

```
- Allocate thread and task queue**

```

/* Allocate thread and task queue */
pool->threads = (pthread_t *) malloc(sizeof(pthread_t) * thread_count);
pool->queue = (threadpool_task_t *) malloc(sizeof(threadpool_task_t) * queue_size);

```

43

Operating Systems 2022

ThreadPool Implementation (cont.)

```

static void *threadpool_thread(void *threadpool)
{
    threadpool_t *pool = (threadpool_t *)threadpool;
    threadpool_task_t task;

    for(;;) {
        /* Lock must be taken to wait on conditional variable */
        pthread_mutex_lock(&(pool->lock));

        /* Wait on condition variable, check for spurious wakeups.
         When returning from pthread_cond_wait(), we own the lock. */
        while((pool->count == 0) && (!pool->shutdown)) {
            pthread_cond_wait(&(pool->notify), &(pool->lock));
        }
    }
}

```

44

43

44

ThreadPool Implementation (cont.)

```

/* Grab our task */
task.function = pool->queue[pool->head].function;
task.argument = pool->queue[pool->head].argument;
pool->head += 1;
pool->head = (pool->head == pool->queue_size) ? 0 : pool->head;
pool->count -= 1;

/* Unlock */
pthread_mutex_unlock(&(pool->lock));

/* Get to work */
(*task.function)(task.argument);
}

```

45

Synchronization Hardware

46

Hardware Support

- The CS problem occurs because the modification of a shared variable may be **interrupted**
- If disable interrupts when in CS
 - Not feasible in multiprocessor machine
 - Clock interrupts cannot fire in any machine
- HW support solution: atomic instructions**
 - atomic: as one **uninterruptible** unit
 - Example: **TestAndSet(var)** and **Swap(a, b)**

47

Atomic TestAndSet()

```

bool TestAndSet (bool &lock) {
    bool value = lock;
    lock = true;
    return value;
}

```

execute atomically:
return the value of "lock" and
set "lock" to true

```

shared data: bool lock; // initially lock = false
// P0
do {
    while (TestAndSet (lock));
    lock = false;
    remainder section
} while (1);

```

```

// P1
do {
    while (TestAndSet (lock));
    lock = false;
    remainder section
} while (1);

```

mutual exclusion? **Y** progress? **Y** bounded-wait? **N**

48

Atomic Swap()

Enter CS if lock == false

shared data: bool lock; // initially lock = false

// P₀
do {

```
key0 = true;
while (key0 == true)
    Swap(lock, key0);
```

```
critical section
lock = false;
remainder section
```

} while (1);

// P₁
do {

```
key1 = true;
while (key1 == true)
    Swap(lock, key1);
```

```
critical section
lock = false;
remainder section
```

} while (1);

mutual exclusion? **Y** progress? **Y** bounded-wait? **N**

49

Atomic CompareAndSwap()

```
int CompareAndSwap (int &value, int expected, int new_value) {
    int temp = value;
    if (value == expected)
        value = new_value;
    return temp;
}
```

shared data: int lock; // initially lock = 0

// P₀
do {

```
while (CompareAndSwap (lock, 0, 1) != 0);
```

```
critical section
```

```
lock = 0;
```

```
remainder section
```

} while (1);

mutual exclusion? **Y** progress? **Y** bounded-wait? **N**

50

Atomic Variables

- **Atomic variable** is another tool that provides **atomic (uninterruptible)** updates on basic data types such as integers and Booleans
- Usually built with atomic instructions such as **CompareAndSwap**
- Example:
 - Let **sequence** be an atomic variable
 - Let **increment()** be an operation for incrementing the atomic variable **sequence**
 - The command **increment(&sequence)** ensures **sequence** is incremented without interruption

51

Atomic Variables (cont.)

- The **increment()** function can be implemented as follows

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

```
bool CompareAndSwap (int &value, int expected, int new_value) {
    int temp = value;
    if (value == expected)
        value = new_value;
    return temp;
}
```

52

Atomic Variables (cont.)

- The **increment()** function can be implemented as follows

```

    bool increment (atomic_int &v) {
        int temp;
        do {
            temp = v;
        } while (temp != (CompareAndSwap (v, temp, temp+1)));
    }
    int CompareAndSwap (int &value, int expected, int new_value) {
        int temp = value;
        if (value == expected)
            value = new_value;
        return temp;
    }
  
```

Annotations: **5** (above temp = v), **5** (above temp !=), **3** (above v), **5** (above temp), **6** (above temp+1), **3** (above value), **5** (above expected), **6** (above new_value). A blue arrow points to the **while** loop.

53

Atomic Variables (cont.)

- The **increment()** function can be implemented as follows

```

    bool increment (atomic_int &v) {
        int temp;
        do {
            temp = v;
        } while (temp != (CompareAndSwap (v, temp, temp+1)));
    }
    int CompareAndSwap (int &value, int expected, int new_value) {
        int temp = value;
        if (value == expected)
            value = new_value;
        return temp;
    }
  
```

Annotations: **3** (above temp = v), **3** (above temp !=), **3** (above v), **3** (above temp), **4** (above temp+1), **3** (above value), **3** (above expected), **4** (above new_value). A blue arrow points to the **while** loop.

54

53

54

Semaphores

55

55

Semaphores

- A tool to generalize the synchronization problem
- More specifically
 - A **record** of **how many units of a particular resource is available**
 - If # record = 1 → binary semaphore, **mutex lock**
 - If # record > 1 → counting semaphore
 - Accessed only through 2 **atomic** operations: **wait** & **signal**
- Spinlock** implementation
 - Semaphore **S** is an integer variable

```

wait (S) {
    while (S <= 0);
    S--;
}
signal (S) {
    S++;
}
  
```

Annotation: **busy waiting** (below the while loop in wait).

56

56

POSIX Semaphore

- Semaphore is part of **POSIX standard** BUT it is not belonged to pthread
 - It can be used with or without thread
- POSIX Semaphore routines

```
#include <semaphore.h>
sem_t sem;
sem_init(&sem);
sem_wait(&sem);
critical section
sem_signal(&sem);
sem_destroy(&sem);
```

57

57

n-Process CS Problem Revisit

- Shared data:


```
semaphore mutex; // initially mutex = 1
```
- Process P_i :


```
do {
    wait(mutex); // pthread_mutex_lock(&mutex)
    critical section
    signal(mutex); // pthread_mutex_unlock(&mutex)
    remainder section
} while (1);
```

progress? **Y**

bounded-wait? **depends on the implementation of wait()**

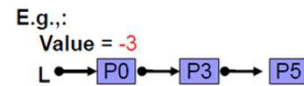
58

58

Semaphores with Non-busy Waiting

- Semaphore is a **data structure with queue**
 - May use any queuing strategy (FIFO, FILO, etc)

```
typedef struct {
    int value; // init to 0
    struct process *L; // PCB queue
} semaphore;
```



- wait()** and **signal()**

- Use system calls: **block()** and **wakeup()**

- Must be **executed atomically**

```
void wait (semaphore S) {
    S.value--; // subtract first
    if (S.value < 0) {
        add this process to S.L;
        sleep();
    }
}

void signal (semaphore S) {
    S.value++;
    if (S.value <= 0) {
        remove this process from S.L;
        wakeup(P);
    }
}
```

59

59

How to Ensure Atomic Wait & Signal Ops?

- Hardware support
 - TestAndSet
 - Swap
- Software solution
 - Peterson's solution
 - Bakery algorithm

60

60

Semaphore with Critical Section

```

void wait (semaphore S) {
    entry section
    S.value--;
    if (S.value < 0) {
        add this process to S.L;
    }
    exit section
    sleep();
} else {
    exit section
}

void signal (semaphore S) {
    entry section
    S.value++;
    if (S.value <= 0) {
        remove this process from S.L;
        wakeup(P);
    }
    exit section
}

```

61

Cooperation Synchronization

- P1 executes S1; P2 executes S2
 - S2 will be executed only after ...
- Implementation
 - Shared variable:
 - semaphore sync; // initially sync = 0

P1:

S1;
signal (sync);

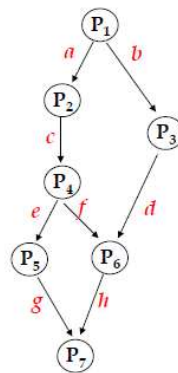
P2:

wait (sync);
S2;

62

A More Complicated Example

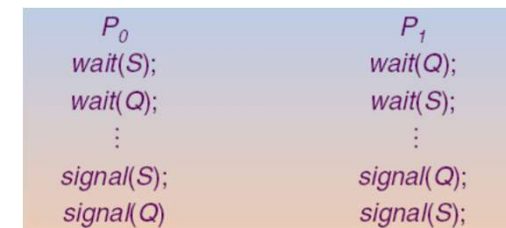
- Initially, all semaphores are 0
- Begin
 - P1: S1; signal(a); signal(b);
 - P2: wait(a); S2; signal(c);
 - P3: wait(b); S3; signal(d);
 - P4: wait(c); S4; signal(e); signal(f);
 - P5: wait(e); S5; signal(g);
 - P6: wait(f); wait(d); S6; signal(h);
 - P7: wait(g); wait(h); S7;
- End



63

Deadlocks and Starvation

- Deadlock**
 - Two processes are waiting indefinitely for each other to release resources
- Starvation**
 - Some processes (threads) wait infinitely



64