Synchronization

CSE 2431: Introduction to Operating Systems
Reading: Chapter 5, [OSC]
(except Section 5.10)
Outline

• Critical region and mutual exclusion
• Mutual exclusion using busy waiting
• Sleep and Wakeup
• Semaphores
• Monitor
• Barrier
• Classic Synchronization Problems
Spooling Example: No Races

Process 1

int next_free;

1. next_free = in;

Stores F1 into next_free;

3. in = next_free + 1;

Process 2

int next_free;

4. next_free = in

5. Stores F2 into next_free;

6. in = next_free + 1;
Spooling Example: Races

Process 1

int next_free;

1 next_free = in;

3 Stores F1 into next_free;

4 in = next_free + 1;

Shared memory

<table>
<thead>
<tr>
<th>4</th>
<th>abc</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Prog.c</td>
</tr>
<tr>
<td>6</td>
<td>Prog.n</td>
</tr>
<tr>
<td>7</td>
<td>F2</td>
</tr>
</tbody>
</table>

Process 2

int next_free;

2 next_free = in;

/* value: 7 */

5 Stores F2 into next_free;

6 in = next_free + 1;
Critical Section (Thread/Process)

- $N$ threads/processes all competing to use the same shared data
- **Race condition** is a situation where two or more threads/processes are reading or writing same shared data and the final result depends on who runs precisely when
- Each thread/process has a code segment, called a **critical section**, in which shared data is accessed
- We need to ensure that when one thread/process is executing in its critical section, **no other thread/process is allowed to execute in its critical section**
Critical Region (Critical Section)

Process {
    while (true) {
        ENTER CRITICAL SECTION
        Access shared variables; // Critical Section;
        LEAVE CRITICAL SECTION
        Do other work
    }
}
Critical Section Characteristics

• **Mutual Exclusion**
  - No other process must execute within its own critical section while a process is in it.

• **Progress**
  - If no process is waiting in its critical section and several processes are trying to get into their critical sections, then entry to the critical section cannot be postponed indefinitely.
    • No process running outside its critical section may block other processes

• **Bounded Wait**
  - A process requesting entry to a critical section should only have to wait for a bounded number of other processes to enter and leave the critical section.
    • No process should have to wait forever to enter its critical section

• **Speed and Number of CPUs**
  - No assumption may be made about speeds or number of CPUs
Critical Regions

Mutual exclusion using critical regions
Outline

• Critical region and mutual exclusion
• Mutual exclusion using busy waiting
• Sleep and Wakeup
• Semaphores
• Monitor
• Barrier
• Classic Synchronization Problems
Synchronization With Busy Waiting

• Possible Solutions
  – Disabling Interrupts
  – Lock Variables
  – Strict Alternation
  – Peterson’s solution
  – TSL
  – Sleep and Wakeup
Disabling Interrupts

• How does it work?
  – Disable all interrupts just before entering a critical section and re-enable them just after leaving it.

• Why does it work?
  – With interrupts disabled, no clock interrupts can occur. (The CPU is only switched from one process to another as a result of clock or other interrupts, and with interrupts disabled, no switching can occur.)

• Problems:
  – What if the process forgets to enable the interrupts?
  – Multiprocessor? (disabling interrupts only affects one CPU)
  – Only used inside OS kernels
int lock;
lock=0

while (lock);
lock = 1;
    Access shared variable; // Critical Section
lock = 0;

Does the above code work?
Strict Alternation

Thread Me; /* For two threads */
{
    while (true)
    {
        while (turn != my_thread_id) {
        Access shared variables; // Critical Section;
        turn = other_thread_id;
        Do other work
    }
}

Satisfies mutual exclusion but not progress.
Why?

• Notes:
  – While {turn != my_thread_id} {}; /* busy waiting */
  – A lock (variable turn) that uses busy waiting is called a spin lock
int flag[2] = {false, false};
Thread Me;
{
    while (true)
    {
        flag[my_thread_id] = true;
        while (flag[other_thread_id]) { };
        Access shared variables; // Critical Section;
        flag[my_thread_id] = false;
        Do other work
    }
}

Can block indefinitely. Why?
Peterson’s Solution

```c
int flag[2]={false, false};
int turn;
Thread Me;
{
    while (true)
    {
        flag[my_thread_id] = true;
        turn = other_thread_id;
        while (flag[other_thread_id] && turn == other_thread_id) { }; // Critical Section;
        Access shared variables;
        flag[my_thread_id] = false;
        Do other work
    }
}
```

It works!!! Why?
Test & Set (TSL)

- Requires hardware support
- Does test and set atomically

```c
char Test_and_Set (char* target);
// All done atomically
{
    char temp = *target;
    *target = true;
    return(temp);
}
```
TSL instruction

enter_region:
   TSL REGISTER, LOCK | copy lock to register and set lock to 1
   CMP REGISTER, #0   | was lock zero?
   JNE enter_region   | if it was non zero, lock was set, so loop
   RET | return to caller; critical region entered

leave_region:
   MOVE LOCK, #0      | store a 0 in lock
   RET | return to caller
Other Similar Hardware Instruction

• **Swap = TSL**

```c
void Swap (char* x,* y);
// All done atomically
{
    char temp = *x;
    *x = *y;
    *y = temp
}
```
Outline

• Critical region and mutual exclusion
• Mutual exclusion using busy waiting
• **Sleep and Wakeup**
• Semaphores
• Monitor
• Barrier
• Classic Synchronization Problems
Sleep and Wakeup

• Problem with previous solutions
  – Busy waiting
  – Wasting CPU
  – Priority Inversion:
    • A high priority waits for a low priority to leave the critical section
    • The low priority can never execute since the high priority is not blocked.

• Solution: sleep and wakeup
  – When blocked, go to sleep
  – Wakeup when it is OK to retry entering the critical section
  – Semaphore operation that executes sleep and wakeup
Types of Syncs in Linux Kernel

- Atomic operation
  - Description: Atomic read-modify-write instructions to a counter; Scope: all CPUs
- Spin lock
  - Description: Lock with busy waiting; Scope: all CPUs
- Semaphore
  - Description: Lock with blocking (wait) sleep; Scope: All CPUs
- Local interrupt disabling
  - Description: Forbid interrupt handling on a single CPU; Scope: Local CPU
- Global interrupt disabling
  - Description: Forbid interrupt handling on all CPUs; Scope: All CPUs
Review: Critical Regions

• What are Data Races
• Critical region and mutual exclusion
• Synchronization using busy waiting
  – Disabling Interrupts
  – Lock Variables
  – Strict Alternation
  – Peterson’s solution
  – TSL
Sleep and Wakeup

• Problem with previous solutions
  – Busy waiting
  – Wasting CPU
  – Priority Inversion:
    • A high priority waits for a low priority to leave the critical section
    • The low priority can never execute since the high priority is not blocked.

• Solution: sleep and wakeup
  – When blocked, go to sleep
  – Wakeup when it is OK to retry entering the critical section
Producer-Consumer Problem (Works?)

```c
#define N 100
int count = 0;  /* number of slots in the buffer */

void producer(void)
{
    int item;

    while (TRUE) { /* repeat forever */
        item = produce_item();  /* generate next item */
        if (count == N) sleep();  /* if buffer is full, go to sleep */
        insert_item(item);
        count = count + 1;  /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);  /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) { /* repeat forever */
        if (count == 0) sleep();  /* if buffer is empty, got to sleep */
        item = remove_item();  /* take item out of buffer */
        count = count - 1;  /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer);  /* was buffer full? */
        consume_item(item);  /* print item */
    }
}
```
Outline

• Critical region and mutual exclusion
• Mutual exclusion using busy waiting
• Sleep and Wakeup
• Semaphores
• Monitor
• Barrier
• Classic Synchronization Problems
Semaphores

• A semaphore count represents the number of abstract resources.

• New variable having two operations
  – The \textit{Wait} (or \textit{Down}, \textit{P}) operation is used to acquire a resource and decrements count.
  – The \textit{Signal} (or \textit{Up}, \textit{V}) operation is used to release a resource and increments count.

• Any semaphore operation is indivisible (\textit{atomic})
Wait and Signal

Wait(S) {
    // or Down(S), P(S)
    S->value --;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

Signal(S) {
    // or Up(S), V(S)
    S->value ++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup (P);
    }
}

- Counting semaphores: 0, ..., N
- Binary semaphores: 0, 1
Counting Semaphores

Semaphore sem = 2 0

P1
Wait(sem);
Display();
Signal(sem);

P2
Wait(sem);
Display();
Signal(sem);

P3
Wait(sem);
Display();
Signal(sem);

● ● ● ● ●
Binary Semaphores

Semaphore sem = 1 0

P1
Wait(sem);
Display();
Signal(sem);

P2
Wait(sem);
Display();
Signal(sem);
Mutex: Binary Semaphore

• Variable with only two states
  – lock
  – unlock

• Mutex is used for mutual exclusion
  – Can be implemented using TSL
  – Can be a specialization of Semaphore (Simplified version of semaphore)
Mutex Implementation using TSL (1)

• Using Test_and_Set (TSL) instruction to implement
  – Mutex_lock:
    • Set lock =1
  – Mutex_unlock
    • Set lock =0
Mutex Implementation Using TSL (2)

mutex_lock:
    TSL REGISTER, MUTEX | copy mutex to register and set mutex to 1
    CMP REGISTER, #0    | was mutex zero?
    JZE ok              | if it was zero, mutex was unlocked, so return
    CALL thread_yield   | mutex is busy; schedule another thread
    JMP mutex_lock      | try again later

ok: RET | return to caller; critical region entered

mutex_unlock:
    MOVE MUTEX, #0      | store a 0 in mutex
    RET | return to caller

Implementation of mutex_lock and mutex_unlock
Producer-Consumer Problem using Semaphores

```c
semaphore mutex = 1; /* controls access to critical region */
semaphore empty = N; /* counts empty buffer slots */
semaphore full = 0; /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item(); /* TRUE is the constant 1 */
        down(&empty); /* generate something to put in buffer */
        down(&mutex); /* decrement empty count */
        insert_item(item); /* enter critical region */
        up(&mutex); /* put new item in buffer */
        up(&full); /* leave critical region */
        up(&mutex); /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full); /* infinite loop */
        down(&mutex); /* decrement full count */
        item = remove_item(); /* enter critical region */
        up(&mutex); /* take item from buffer */
        up(&empty); /* leave critical region */
        up(&mutex); /* increment count of empty slots */
        consume_item(item); /* do something with the item */
    }
}
```
Using Mutex to Implement Semaphores

• Using `mutex_lock` and `mutex_unlock` to implement a counter semaphore
  – Wait (or Up, P)
  – Signal (or Down, V)
class Semaphore {
    Mutex m;    // Mutual exclusion.
    int count;     // Resource count.

public:
    Semaphore(int count );
    void Wait();
    void Signal();

};

static inline Semaphore::Semaphore( int count )
{
    count = count;
}
void Semaphore::Wait()
{
    mutex_lock(m);
    while (count == 0)
    {
        mutex_unlock(m);
        yield();
        mutex_lock(m);
    }
    count--;
    mutex_unlock(m);
}

void Semaphore::Signal()
{
    mutex_lock(m);
    count++;
    mutex_unlock(m);
}
Semaphore Implementation Using Sleep & Wakeup

typedef struct
{
    int value;
    struct process *list;
} Semaphore;

Wait(Semaphore *S)
{
    S->value = S->value - 1;
    if (S->value < 0)
    {
        add this process to S->list;
        block();
    }
}

Signal(Semaphore *S)
{
    S->value = S->value + 1;
    if (S->value <= 0)
    {
        remove a process P from S->list;
        wakeup(P);
    }
}

• Skipped locks here in order to provide atomicity
Tradeoffs

• Busy waiting (spinlock)
  – Waste CPU cycles

• Sleep & Wakeup (blocked lock)
  – Context switch overhead

• Hybrid competitive solution
  – Apply spinlocks if the waiting time is shorter than the context switch time
  – Use sleep & wakeup if the waiting time is longer than the context switch time
  – Why?
  – What if you don’t know the waiting time?
Possible Deadlocks with Semaphores

Example:

P0
share two semaphores S and Q
S:=1; Q:=1;

Wait(S); // S=0 ------------> Wait(Q); // Q=0
Wait(Q); // Q=-1  <-------->  Wait(S); // S=-1

// P0 blocked

P1

// P1 blocked

DEADLOCK

Signal(S);
Signal(Q);
Signal(Q);
Signal(S);
Be Careful When Using Semaphores

// Violation of Mutual Exclusion
Signal(mutex);           mutexUnlock();
critical section         criticalSection();
Wait(mutex);             mutexLock();

// Deadlock Situation
Wait(mutex);            mutexLock(P);
critical section        criticalSection();
Wait(mutex);            mutexLock(P);

// Violation of Mutual Exclusion (omit wait(mutex)/mutexLock())
critical section        critical Section();
Signal(mutex);           mutexUnlock();

// Deadlock Situation (omit signal(mutex)/mutexUnlock())
Wait(mutex);            mutexLock();
critical section        criticalSection();
Outline

• Critical region and mutual exclusion
• Mutual exclusion using busy waiting
• Sleep and Wakeup
• Semaphores
• Monitors
• Barrier
• Classic Synchronization Problems
Monitors

• A simpler way to synchronize
• A set of programmer-defined operators

```
monitor monitor-name
{

    // variable declaration

    public entry P1(..);
    { ... };

    .......

    public entry Pn(..);
    { ... };

    begin
        initialization code
    end
}
```
Monitor Properties

• The internal implementation of a monitor type cannot be accessed directly by various threads.
• The encapsulation provided by the monitor type limits access to the local variables only by the local procedures.
• Monitor construct does not allow concurrent access to all procedures defined within the monitor.
• Only one thread/process can be active within the monitor at a time.
• Synchronization is built-in.
Outline

• Critical region and mutual exclusion
• Mutual exclusion using busy waiting
• Sleep and Wakeup
• Semaphores
• Monitor
• Barrier
• Classic Synchronization Problems
Barriers

• Use of a barrier
  – Processes approaching a barrier
  – All processes but one blocked at barrier
  – Last process arrives, all are let through

• Problem:
  – Wastes CPU if workloads are unbalanced
Barriers
How to Implement a Barrier?

• For $N$ processes: using messages
• For $N$ threads: using shared variables
Outline

• Critical region and mutual exclusion
• Mutual exclusion using busy waiting
• Sleep and Wakeup
• Semaphores
• Monitor
• Barrier
• Classic Synchronization Problems
Classic Problems

• Bounded buffer problem ✓
• Reader-writer problem ✓
• Dining philosophers problem ✓
• Sleeping barber
Bounded Buffer Problem (1)

• A producer: in an infinite loop and produce one item each iteration into the buffer
• A consumer: in an infinite look and consumes one item each iteration from the buffer
• Buffer size: can only hold at most $N$ items
Bounded Buffer Problem (2)

Semaphore mutex; // shared and initialized to 1
Semaphore empty; // counts empty buffers, initialized to n
Semaphore full; // counts full buffers, initialized to 0

// Producer
repeat
    ....
    produce an item in nextp
    ....
    Wait(empty);
    Wait(mutex);
    nextc
    ....
    add nextp to buffer
    ....
    Signal(mutex);
    Signal(full);
    ..... until false;

// Consumer
repeat
    ....
    Wait(full);
    Wait(mutex);
    ....
    remove an item from buffer to
    ....
    Signal(mutex);
    Signal(empty);
    ....
    consume the item in nextc
    ..... until false;
Reader-Writer Problem

- Readers: read data
- Writers: write data
- Rule:
  - Multiple readers can read the data simultaneously
  - Only one writer can write the data at any time
  - A reader and a writer cannot in critical section concurrently.
- Locking table: whether any two can be in the critical section simultaneously

<table>
<thead>
<tr>
<th></th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reader</td>
<td><strong>OK</strong></td>
<td>No</td>
</tr>
<tr>
<td>Writer</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Reader-Writer Solution

• Does it work? Why?
• What if?
• Problem with this solution?

Semaphore mutex, wrt; // shared and initialized to 1;
int readcount; // shared and initialized to 0

// Writer
wait(mutex);
readcount:=readcount+1;
wait(wrt); if readcount == 1 then wait(wrt);
writing performed
......

// Reader
wait(mutex);
readcount:=readcount+1;
if readcount == 1 then wait(wrt);
signal(mutex);
reading performed
......
signal(wrt);
readcount:=readcount-1;
if readcount == 0 then signal(wrt);
signal(mutex);
Dining Philosophers Problem

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- Possible deadlock?
- How to prevent deadlock?
Solution to Dining Philosophers Problem?

```c
#define N 5

void philosopher(int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1) % N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}

NOT a solution to the dining philosophers problem
```
Dining Philosophers Solution (1)

```c
#define N 5 /* number of philosophers */
#define LEFT (i+N-1)%N /* number of i's left neighbor */
#define RIGHT (i+1)%N /* number of i's right neighbor */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is trying to get forks */
#define EATING 2 /* philosopher is eating */

typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE) {
        think(); /* repeat forever */
        take_forks(i); /* philosopher is thinking */
        acquire two forks or block */
        eat(); /* yum-yum, spaghetti */
        put_forks(i); /* put both forks back on table */
    }
}
```

How to implement `take_forks()` and `put_forks()`?
Dining Philosophers Solution (2)

void take_forks(int i)  // i: philosopher number, from 0 to N−1 */
{
    down(&mutex);       // enter critical region */
    state[i] = HUNGRY;  // record fact that philosopher i is hungry */
    test(i);            // try to acquire 2 forks */
    up(&mutex);         // exit critical region */
    down(&s[i]);        // block if forks were not acquired */
}

void put_forks(i)       // i: philosopher number, from 0 to N−1 */
{
    down(&mutex);       // enter critical region */
    state[i] = THINKING;  // philosopher has finished eating */
    test(LEFT);         // see if left neighbor can now eat */
    test(RIGHT);        // see if right neighbor can now eat */
    up(&mutex);         // exit critical region */
}

void test(i)            // i: philosopher number, from 0 to N−1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
    {
        state[i] = EATING;
        up(&s[i]);
    }
}
Dining Philosophers Solution (3)

```c
void take_forks(int i) {
    down(&mutex);
    state[i] = HUNGRY;
test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i) {
    down(&mutex);
    state[i] = THINKING;
test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```
Dining Philosophers Solution (4)

```c
void take_forks(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i)
    / i: philosopher number, from 0 to N–1 */
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i) {
    / i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```
Dining Philosophers Solution (5)

```c
void take_forks(int i) {
    down(&mutex); /* enter critical region */
    state[i] = HUNGRY; /* record fact that philosopher i is hungry */
    test(i); /* try to acquire 2 forks */
    up(&mutex); /* exit critical region */
    down(&s[i]); /* block if forks were not acquired */
}

void put_forks(i) {
    down(&mutex); /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT); /* see if left neighbor can now eat */
    test(RIGHT); /* see if right neighbor can now eat */
    up(&mutex); /* exit critical region */
}

void test(i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```
The Sleeping Barber Problem

- $N$ customer Chairs
- One barber can cut one customer’s hair at any time
- No customer, goes to sleep
The Sleeping Barber Solution (1)

#define CHAIRS 5 /* # chairs for waiting customers */
typedef int semaphore; /* use your imagination */
semaphore customers = 0; /* # of customers waiting for service */
semaphore barbers = 0; /* # of barbers waiting for customers */
semaphore mutex = 1; /* for mutual exclusion */
int waiting = 0; /* customers are waiting (not being cut) */
The Sleeping Barber Solution (2)

```c
void barber(void) {
    while (TRUE) {
        down(&customers); /* go to sleep if # of customers is 0 */
        down(&mutex);    /* acquire access to 'waiting' */
        waiting = waiting - 1; /* decrement count of waiting customers */
        up(&barbers);     /* one barber is now ready to cut hair */
        up(&mutex);       /* release 'waiting' */
        cut_hair();       /* cut hair (outside critical region) */
    }
}
```
void customer(void)
{
    down(&mutex);  /* enter critical region */
    if (waiting < CHAIRS) {
        waiting = waiting + 1;  /* if there are no free chairs, leave */
        up(&customers);  /* increment count of waiting customers */
        up(&mutex);  /* wake up barber if necessary */
        up(&mutex);  /* release access to 'waiting' */
        down(&barbers);  /* go to sleep if # of free barbers is 0 */
        get_haircut();  /* be seated and be serviced */
    } else {
        up(&mutex);  /* shop is full; do not wait */
    }
}
Summary (I)

• Critical region and mutual exclusion
• Mutual exclusion using busy waiting
  – Disabling Interrupts
  – Lock Variables
  – Strict Alternation
  – Peterson’s solution
  – TSL
  – Sleep and Wakeup
Summary (II)

- Sleep and Wakeup
- Semaphores
- Monitor
- Barrier
Summary/Lessons Learned (III)

- Synchronization is very important in OS when accessing kernel data structures.
- System performance may vary considerably, depending on the kind of synchronization primitive selected.
- **Rule of thumb adopted by kernel developers:**
  - Always keep the concurrency level as high as possible in the system.
- Concurrency level in the system depends on two factors:
  - Number of I/O devices that operate concurrently.
  - Number of CPUs that do productive work.
- To maximize I/O throughput, interrupts should be disabled for very short time.
- To use CPUs efficiently, sync primitives based on spin locks should be avoided whenever possible.
- Generally speaking, choosing the sync primitives depends on what kinds of kernel control paths access the data structure.