Synchronization
Introducing: Critical Region
(Critical Section)

Process {
    while (true) {
        ENTER CRITICAL REGION
        Access shared variables;
        LEAVE CRITICAL REGION
        Do other work
    }
}
Critical Region Requirements

- Mutual Exclusion
  - Safety
- Progress
  - No deadlock
- Bounded Wait
  - No starvation
Critical Regions

Mutual exclusion using critical regions

What mechanisms do we need to be able to achieve mutual exclusion?

A way to block B

A way to let B know that it can proceed
Mutual Exclusion Solutions

- Software-only candidate solutions (Two-Process Solutions)
  - Lock Variables
  - Turn Mutual Exclusion
  - Other Flag Mutual Exclusion
  - Two Flag Mutual Exclusion
  - Two Flag and Turn Mutual Exclusion

- Hardware solutions
  - Disabling Interrupts; Test-and-set; Swap (Exchange)

- Semaphores
... 
while (lock) {
    /* spin spin spin spin spin */
}
lock = 1;

/* EnterCriticalSection; */
access shared variable;
/* LeaveCriticalSection; */
lock = 0;
...

What's the problem?
Turn-based Mutual Exclusion with Strict Alternation

... 

while (turn != my_process_id) {
    /* wait your turn */
}

access shared variables;

turn = other_process_id;

...

What's the problem?
Other Flag Mutual Exclusion

```c
int owner[2] = {false, false};
...
while (owner[other_process_id]) {
    /* wait your turn */
}
owner[my_process_id] = true;
access shared variables;
owner[my_process_id] = false;
...
```

What's the problem?
int owner[2] = {false, false};
...
owner[my_process_id] = true;
while (owner[other_process_id]) {
    /* wait your turn */
}

access shared variables;
owner[my_process_id] = false;
...

What's the problem?
Two Flag and Turn Mutual Exclusion

```c
int owner[2]={false, false};
int turn;
...
owner[my_process_id] = true;
turn = other_process_id;
while (owner[other_process_id] and
       turn == other_process_id) {
    /* wait your turn */
}
access shared variables;
owner[my_process_id] = false;
...
Discussion

- In uni-processors
  - Concurrent processes cannot be overlapped, only **interleaved**
  - A process runs until it **invokes a system call**, or is **interrupted**
  - To guarantee mutual exclusion, **hardware support** could help by allowing the disabling of interrupts
    
    ```c
    While(true) {
        /* disable interrupts */
        /* critical section */
        /* enable interrupts */
        /* remainder */
    }
    ```
  - What’s the problem with this solution?
Discussion

- In multi-processors
  - Several processors share memory
  - Processors behave independently in a peer relationship
  - Interrupt disabling will not work
  - We need **hardware support** where access to a memory location excludes any other access to that same location
  - The hardware support is based on execution of multiple instructions **atomically** (test and set)
boolean Test_And_Set(boolean* lock) {
  atomic {
    boolean initial;
    initial = *lock;
    *lock = true;
    return initial;
  }
  atomic = executed in a single shot without any interruption

Note: this is more accurate than the textbook version
Using Test_And_Set for Mutual Exclusion

\[ P_i \{ \]
\[ \quad \text{while}(1) \{ \]
\[ \quad \quad \text{while}(\text{Test\_And\_Set}(\text{lock})) \{ \]
\[ \quad \quad \quad \text{/* spin */} \]
\[ \quad \}
\[ \}
\[ /* \text{Critical Section} */ \]
\[ lock = 0; \]
\[ /* \text{remainder} */ \]
\[ \}
\[ \}

\[ \]

void main () {
    lock = 0;
    parbegin(P_1, \ldots, P_n);
}

What's the problem?
Semaphores

Fundamental Principle:
- Two or more processes want to cooperate by means of simple signals

Special Variable: semaphore s
- A special kind of “int” variable
- Can’t just modify or set or increment or decrement it
Semaphores

- Before entering critical section
  - `semWait(s)`
    - Receive signal via semaphore `s`
    - “down” on the semaphore
    - Also: `P` – proberen

- After finishing critical section
  - `semSignal(s)`
    - Transmit signal via semaphore `s`
    - “up” on the semaphore
    - Also: `V` – verhogen

- Implementation requirements
  - `semSignal` and `semWait` must be atomic
### Semaphores vs. Test_and_Set

#### Semaphore

```c
semaphore s = 1;
P_i {
    while(1) {
        semWait(s);
        /* Critical Section */
        semSignal(s);
        /* remainder */
    }
}
```

#### Test_and_Set

```c
lock = 0;
P_i {
    while(1) {
        while(Test_And_Set(lock));
        /* Critical Section */
        lock =0;
        /* remainder */
    }
}
```

- Avoid busy waiting by suspending
  - Block if $s == \text{False}$
  - Wakeup on signal ($s = \text{True}$)
Inside a Semaphore

- **Requirement**
  - No two processes can execute `wait()` and `signal()` on the same semaphore at the same time!

- **Critical section**
  - `wait()` and `signal()` code
  - Now have busy waiting in critical section implementation
    - Implementation code is short
    - Little busy waiting if critical section rarely occupied
    - Bad for applications may spend lots of time in critical sections
Inside a Semaphore

- Add a waiting queue
- Multiple process waiting on $s$
  - Wake up one of the blocked processes upon getting a signal

Semaphore data structure

```c
typedef struct {
    int count;
    queueType queue;
    /* queue for procs. waiting on s */
} SEMAPHORE;
```
typedef struct bsemaphore {
    enum {0,1} value;
    queueType queue;
} BSEMAPHORE;

void semSignalB(bsemaphore s) {
    if (s.queue is empty())
        s.value = 1;
    else {
        remove P from s.queue;
        place P on ready list;
    }
}

void semWaitB(bsemaphore s) {
    if (s.value == 1)
        s.value = 0;
    else {
        place P in s.queue;
        block P;
    }
}
typedef struct {
    int count;
    queueType queue;
} SEMAPHORE;

void semSignal(semaphore s) {
    s.count++;
    if (s.count ≤ 0) {
        remove P from s.queue;
        place P on ready list;
    }
}

void semWait(semaphore s) {
    s.count--;
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }
}
Isn’t this exactly what semaphores were trying to solve? Are we stuck?!
Solution: resort to test-and-set

```c
typedef struct {
    boolean lock;
    int count;
    queueType queue;
} SEMAPHORE;

void semWait(semaphore s) {
    while (test_and_set(lock)) { }
    s.count--;
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }
    lock = 0;
}
```
Making the operations atomic

- Busy-waiting again!
- Then how are semaphores better than just using `test_and_set`?

```c
void semWait(semaphore s) {
    while (test_and_set(lock)) {} 
    s.count--; 
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }
    lock = 0;
}
```

- T&S: busy-wait during critical section
- Sem.: busy-wait just during `semWait`, `semSignal`: very short operations!
mutual exclusion using semaphores

semaphore s = 1;
Pi {
    while(1) {
        semWait(s);
        /* Critical Section */
        semSignal(s);
        /* remainder */
    }
}

Copyright ©: University of Illinois CS 241 Staff
Value of Semaphore

queue

lock

1

semWait(lock)

0

B

-1

semSignal(lock)

0

1

Process A

Process B

Queue

Critical Region

Normal Execution

Blocked on semaphore lock

Process

Critical Region

Normal Execution

Blocked on semaphore lock
Semaphore Example 1

```
semaphore s = 2;
P_i {
    while(1) {
        semWait(s);
        /* CS */
        semSignal(s);
        /* remainder */
    }
}
```

- What happens?
- When might this be desirable?
Semaphore Example 2

```c
semaphore s = 0;

P_i {
    while(1) {
        semWait(s);
        /* CS */
        semSignal(s);
        /* remainder */
    }
}
```

- What happens?
- When might this be desirable?
Semaphore Example 3

```c
semaphore s = 0;                semaphore s; /* shared */
P1 {                                P2 {
    /* do some stuff */             /* do some stuff */
    semWait(s);                    semSignal(s);
    /* do some more stuff */       /* do some more stuff */
}
```

- What happens?
- When might this be desirable?
Semaphore Example 4

Process 1 executes:

```c
while(1) {
    semWait(S);
    a;
    semSignal(Q);
}
```

Process 2 executes:

```c
while(1) {
    semWait(Q);
    b;
    semSignal(S);
}
```

- Two processes
- Two semaphores: S and Q
- Protect two critical variables ‘a’ and ‘b’.
- What happens in the pseudocode if Semaphores S and Q are initialized to 1 (or 0)?
Summary

- Synchronization is important for correct multi-threading programs
- Critical regions
- Solutions to protect critical regions
  - Software-only approaches
  - Other hardware solutions
  - Semaphores