Synchronization and Semaphores
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
    ```
    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) {
    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

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

void main () {
    lock = 0;
    parbegin(P_1, ..., 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 \( s == False \)
  - Wakeup on signal \( s = True \)
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`
  - Wakeup 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;
```
Binary Semaphores

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;
    }
}

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

- 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;

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

Copyright ©: University of Illinois CS 241 Staff
Queue | Value of Semaphore
---|---
A | lock
0 | 1

Process A

semWait(lock)

Process B

semWait(lock)

-1

semSignal(lock)

0

semSignal(lock)

1

Copyright ©: University of Illinois CS 241 Staff
Semaphore Example 1

```c
Semaphore s = 2;
Pi {
    while(1) {
        semWait(s);
        /* CS */
        semSignal(s);
        /* remainder */
    }
}
```

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

```
semaphore s = 0;
Pi {
    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 {  
    /* do some stuff */  
    semWait(s);  
    /* do some more stuff */
}  
P2 {  
    /* do some stuff */  
    semSignal(s);  
    /* 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(Q);
}
```

- 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)?
Be careful!

Deadlock or Violation of Mutual Exclusion?

1. `semSignal(s);`
2. `semWait(s);`
3. `critical_section();`
4. `semWait(s);`
5. `semWait(s);`
6. `critical_section();`
7. `semSignal(s);`
8. `semSignal(s);`
POSIX Semaphores

- Named Semaphores
  - Provides synchronization between unrelated process and related process as well as between threads
  - Kernel persistence
  - System-wide and limited in number
  - Uses `sem_open`

- Unnamed Semaphores
  - Provides synchronization between threads and between related processes
  - Thread-shared or process-shared
  - Uses `sem_init`
POSIX Semaphores

Data type
- Semaphore is a variable of type `sem_t`

Include `<semaphore.h>`

Atomic Operations

```c
int sem_init(sem_t *sem, int pshared, unsigned value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```
#include <semaphore.h>

```c
int sem_init(sem_t *sem, int pshared, unsigned value);
```

- Initialize an unnamed semaphore
- Returns
  - 0 on success
  - -1 on failure, sets `errno`
- Parameters
  - `sem`:
    - Target semaphore
  - `pshared`:
    - 0: only threads of the creating process can use the semaphore
    - Non-0: other processes can use the semaphore
  - `value`:
    - Initial value of the semaphore

You cannot make a copy of a semaphore variable!!!
Sharing Semaphores

- Sharing semaphores between threads within a process is easy, use `pshared==0`
  - Forking a process creates copies of any semaphore it has… `sem_t` semaphores are not shared across processes

- A non-zero `pshared` allows any process that can access the semaphore to use it
  - Places the semaphore in the global (OS) environment
`sem_init` can fail

- On failure
  - `sem_init` returns -1 and sets `errno`

<table>
<thead>
<tr>
<th><code>errno</code></th>
<th>cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td><code>Value &gt; sem_value_max</code></td>
</tr>
<tr>
<td>ENOSPC</td>
<td>Resources exhausted</td>
</tr>
<tr>
<td>EPERM</td>
<td>Insufficient privileges</td>
</tr>
</tbody>
</table>

```c
sem_t semA;

if (sem_init(&semA, 0, 1) == -1) {
  perror("Failed to initialize semaphore semA");
}
```
Semaphore Operations

```c
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

- **Destroy an semaphore**
- **Returns**
  - 0 on success
  - -1 on failure, sets `errno`
- **Parameters**
  - `sem`:
    - Target semaphore
- **Notes**
  - Can destroy a `sem_t` only once
  - Destroying a destroyed semaphore gives undefined results
  - Destroying a semaphore on which a thread is blocked gives undefined results

Copyright ©: University of Illinois CS 241 Staff
Semaphore Operations

```c
#include <semaphore.h>
int sem_post(sem_t *sem);
```

- Unlock a semaphore
- Returns
  - 0 on success
  - -1 on failure, sets `errno` (== EINVAL if semaphore doesn’t exist)
- Parameters
  - `sem`:
    - Target semaphore
    - `sem > 0`: no threads were blocked on this semaphore, the semaphore value is incremented
    - `sem == 0`: one blocked thread will be allowed to run
- Notes
  - `sem_post()` is reentrant with respect to signals and may be invoked from a signal-catching function
Semaphore Operations

```c
#include <semaphore.h>
int sem_wait(sem_t *sem);
```

- **Lock a semaphore**
  - Blocks if semaphore value is zero
- **Returns**
  - 0 on success
  - -1 on failure, sets `errno` (== EINTR if interrupted by a signal)
- **Parameters**
  - `sem`
    - Target semaphore
    - sem > 0: thread acquires lock
    - sem == 0: thread blocks
Semaphore Operations

```c
#include <semaphore.h>
int sem_trywait(sem_t *sem);
```
- Test a semaphore’s current condition
  - Does not block
- Returns
  - 0 on success
  - -1 on failure, sets `errno` (== AGAIN if semaphore already locked)
- Parameters
  - `sem`:
    - Target semaphore
    - sem > 0: thread acquires lock
    - sem == 0: thread returns
Example: bank balance

- Want shared variable `balance` to be protected by semaphore when used in:
  - `decshared` – a function that decrements the current value of `balance`
  - `incshared` – a function that increments the `balance` variable.
Example: bank balance

```c
int decshared() {
    while (sem_wait(&balance_sem) == -1)
        if (errno != EINTR)
            return -1;
    balance--;
    return sem_post(&balance_sem);
}

int incshared() {
    while (sem_wait(&balance_sem) == -1)
        if (errno != EINTR)
            return -1;
    balance++;
    return sem_post(&balance_sem);
}
```
Summary

- Semaphores
- Semaphore implementation
- POSIX Semaphore
- Programming with semaphores

Next time: solving real problems with semaphores & more