CSCI 4730/6730 OS
(Chap #6 Synchronization Tools – Part 3)

In Kee Kim

Department of Computer Science

University of Georgia

Where are we?

- Software Tools for Synchronization
  - Mutex Locks (Last Class)
  - Semaphores
  - SW-based approach, synchronization tools for application programmers

Semaphore

- **Semaphore**
  - Provides much *nicer* (sophisticated) ways than Mutex locks

- What is Mutex Lock?
  - Boolean variable indicating if lock is ***available*** or **not**
  - Two Operations in Mutex Lock?
    - **`acquire()`**
    - **`release()`**

Recap: `acquire()` and `release()`

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

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

```
while (true) {
        acquire();

            critical section

        release();

        remainder section
}
```

Semaphore

❑ Semaphore

- Provides much *nicer* (sophisticated) ways than Mutex locks
- Basically generalized-version of (mutex) locks
- A special type of variable that supports two atomic (indivisible) operations
- Invented by Dijkstra in 1965

Semaphore

- Semaphore ***S*** – integer variable that can be accessed via two *indivisible* (atomic) operations
  - **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**
  - **`P: wait():`** "to test"
  - **`V: signal():`** "to increment"

Semaphore

- Indivisible Operation
  - Atomic Operation
  - All the modifications to **S (Semaphore)** value in the **`wait()`** and **`signal()`** operations must be executed indivisibly.
  - One process modifies **S** value, **NO other process** can simultaneously modify that same **S** value.

Definition of `wait()` and `signal()`

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

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

Semaphore

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

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

Mutex Lock

Definition of `wait()` and `signal()`

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

0 or less means
someone else is in the critical section

1 or higher means
you can enter into the critical section

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

When you are leaving
You should call `signal(S)`

Semaphore

- **Binary (or Mutex) semaphore**
  - Integer value can range only between 0 and 1
  - Same as mutex lock
- **Counting semaphore**
  - Integer value can range over an unrestricted domain
  - It can be used *when multiple resources are available*
  - It can be also used for *scheduling and queue management*

Semaphore

- Obviously, with semaphore, you can make a solution to Critical Section Problem
  - Create a semaphore "**`mutex`**" initialized to 1

```
wait(mutex);    // mutex--; only I can enter CS
   Critical Section
signal(mutex); // mutex++; anyone can enter CS
```

Semaphore (Coordinated Exec.)

- Consider P_1 and P_2 that with two statements S_1 and S_2.
- Requirement: S_1 needs to be executed before S_2

```
P1:
    S1;
```

```
P2:
    S2;
```

Any idea?

Semaphore (Coordinated Exec.)

- Consider P_1 and P_2 that with two statements S_1 and S_2 .
- Requirement: S_1 needs to be executed before S_2
  - You can create a semaphore "**`synch`**" initialized to 0
    - Note: I don't recommend this coding style (**too ad-hoc**)
  - What does **`synch == 0`** mean?
    - Why we create **`synch`** initialized to 0

```
P1:
    S1;
```

```
P2:
    S2;
```

Semaphore (Coordinated Exec.)

- Consider P_1 and P_2 that with two statements S_1 and S_2.
- Requirement: S_1 needs to be executed before S_2
  - You can create a semaphore "**`synch`**" initialized to 0
    - Note: I don't recommend this coding style (**too ad-hoc**)

```
P1:
    S1;                 // 2) Run S1
    signal(synch);
                        // 3) synch == 1

P2:                     // 1) Hold b/c synch is 0
    wait(synch);
    S2;                 // 4) Now, you can enter and run S2
```

Too Much Milk, Try #6 (Mutex Lock)

Thread A

```
lock.acquire();
if (!milk)
  buy milk
lock.release();
```

Thread B

```
lock.acquire();
if (!milk)
  buy milk
lock.release();
```

Too Much Milk, Try #7 (Semaphore)

Thread A

```
wait(S);
if (!milk)
  buy milk
signal(S);
```

Thread B

```
wait(S);
if (!milk)
  buy milk
signal(S);
```

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same **Semaphore** simultaneously!
  - Ugh -- The semaphore implementation itself becomes the critical section problem ☹
  - **`wait()`** and **`signal()`** code are placed in the critical section

Busy Waiting

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

What is benefit of busy waiting?

What is disadvantage of busy waiting?

- **Busy Waiting** is not generally preferred.
  - Due to high CPU consumption
  - Resource cannot be used for other processes
- **Busy Waiting** is only OK for
  - Scheduling overhead is larger than expected wait time
  - Processor resources are not needed for other tasks

Semaphore Implementation w/o Busy waiting

- Using blocking operation + waiting queue
- Two operations:
  - `block` – place the process invoking the operation on the appropriate waiting queue
  - `wakeup` – remove one of processes in the waiting queue and place it in the ready queue
- You studied scheduling algorithms – what scheduling algorithm can be used for waiting queue?

Semaphore Implementation w/o Busy waiting

<u>Semaphore (initial value = 1)</u>

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

Q. What happens if s→value == 0?

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
```

Q. If (s→value <=0)??? Why not s→value> 0?

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

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


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

← original code
w/ Busy waiting

Problems with Semaphores

- **Deadlock (Chapter 8)** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

```
P0                  P1
wait (S);           wait (Q);
wait (Q);           wait (S);

...                 ...

signal(S);          signal(Q)
signal(Q);          signal(S)
```

Problems with Semaphores

- Error-prone
  - **P**(**`wait`**) and **V**(**`signal`**) must be matched

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

Common error - Omitting signal(mutex) or wait(mutex) or both…

Semaphore Disadvantages

- Semaphores are essentially shared global integer variables
- Access to semaphores can come from anywhere in a program
- Too many Semaphores – because Semaphores can be used for multiple purposes. i.e., mutual exclusion, queue management…
- There is no control mechanism for proper use

Last Thing: Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool. e.g., mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.
  - Poor performance or responsiveness

Liveness

- Two examples.
  1. Deadlock
  2. Priority Inversion
     - Three processes — L, M, and H (Priority H > M> L)
     - H requires semaphore S, which is currently being used by L
     - So H is waiting
     - Now process M becomes runnable and preempts L.
     - L is scheduled before H ☹

End of Chapter 6