Semaphores
Readings

- Silberschatz: Chapter 5
Mutual Exclusion in Critical Sections

A enters critical region
A leaves critical region
B attempts to enter critical region
B enters critical region
B leaves critical region

Process A

Process B

Time

\( T_1 \)
\( T_2 \)
\( T_3 \)
\( T_4 \)
Today there are libraries that provide application programmers with **semaphores**

Semaphores are used by programmers to
- ensure mutual exclusion
- send signals from one process to another

Let’s first talk about semaphores and then the OS support for them.
What is a semaphore?

- A semaphore is an integer variable with the following three operations.
  1. **Initialize**: You can initialize the semaphore to any non-negative value.
  2. **Decrement**: A process can decrement the semaphore, if its value is positive. If value is 0, the process blocks (gets put into queue). It is said to be sleeping on the semaphore. This is the down operation.
  3. **Increment**: If value is 0 and some processes are sleeping on the semaphore, one is unblocked. Otherwise, value is incremented. This is the up operation.
What is a Semaphore?

- Use **down** before entering a critical section
- Use **up** after finishing with a critical section i.e.,
- Example: Assume \( S \) is initialized to 1.

```
........

down(S);
    //critical section
up(S);
```
Using Semaphores

Process $P_0$

\[
\text{down}(S); \\
\quad \text{//critical section} \\
\text{up}(S);
\]

Process $P_1$

\[
\text{down}(S); \\
\quad \text{//critical section} \\
\text{up}(S);
\]
Using Semaphores

Process $P_0$
\[ \text{down}(S); \]
\[
\text{critical section}
\]
\[ \text{up}(S); \]

Process $P_1$
\[ \text{down}(S); \]
\[
\text{critical section}
\]
\[ \text{up}(S); \]

- Initialize the semaphore variable, $S$, to 1
  - Why not zero?
Now what would happen if $P_0$ executes the down operation?

- The semaphore $S$ is currently 1.
- It becomes 0 and $P_0$ enters the critical section.
Using Semaphores

Process $P_0$
\[\text{down}(S);\]
\[\text{critical section}\]
\[\text{up}(S);\]

Process $P_1$
\[\text{down}(S);\]
\[\text{critical section}\]
\[\text{up}(S);\]

- Now what would happen if $P_1$ executes the \textit{down} operation?
  - The semaphore $S$ is currently 0
  - $P_1$ blocks
Using Semaphores

Process $P_0$

```plaintext
down(S);
critical section
up(S);
```

Process $P_1$

```plaintext
down(S);
critical section
up(S);
```

- Assume now that $P_0$ is done with the critical section
- It calls the up function
  - $P_1$ is unblocked
  - If there was no process waiting to enter the critical section the value of $S$ would become one
Using Semaphores

- What happens if there are three processes: $P_0, P_1, P_2$?
- Assume $P_0$ enters its critical section.
- If $P_1$ and $P_2$ execute the \texttt{down} operation, they will block.
- When $P_0$ leaves the critical section, then $P_1$ is unblocked, allowing $P_1$ to enter its critical section.
- $P_2$ is still blocked.
- What if $P_0$ wants to enter its critical section again when $P_1$ is in it?
Using Semaphores

- What if we want to allow 10 processes to use a critical section?
- How would we initialize a semaphore, $S$?
Semaphore

- **Binary Semaphore**: Value is either 0 or 1
  - Often referred to as a *mutex*
- **Counting Semaphore**: Value ranges from 0 to \( N \) where \( N \) can be any integer number
Deadlock

- **Deadlock** - Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Something to watch for
Example: Let $S$ and $Q$ be two semaphores initialized to one.

- **Process $P_0$**
  - `down(S);`
  - `down(Q);`
  - `......`
  - `up(S);`
  - `up(Q);`

- **Process $P_1$**
  - `down(Q);`
  - `down(S);`
  - `......`
  - `up(Q);`
  - `up(S);`
Implementation

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
A semaphore can be defined as a C struct along these lines:

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

- The `down()` operation can be defined as

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

- The `block()` operation suspends the process that invokes it.
Implementation

- **up()** operation can be defined as

```c
up(semaphore *S) {
    S->value++;
    if (S->value ≤ 0) {
        remove process P from S->list;
        wakeup(P);
    }
}
```

- The **wakeup()** operation sends a signal that represents an event that the invoking process is no longer in the critical section.
Implementation

- BTW, the implementation just described is how Linux implements semaphores.
- The `up` and `down` operations represent require access to a **critical section** which is the semaphore variable.
- Need hardware/OS support e.g.,
  - Signals, TSL
  - Signals allow for a “message” to be sent to processes.
Semaphore Implementation

- When the up is executed a blocked process is woken up. This is done using signals.
- Semaphore operations are critical sections - use TSL.
Questions

- How are multiple processes prevented from being in the critical section?

- Why different than disabling interrupts?

- Which is better in a multicore system?
  - Disabling interrupts or TSL test?
Summary

- Defined race condition
- Examined different solutions