**Thread-safe counter**

*Contributed by Benjamin Yang*.

Create a counter that is incremented by two threads, 5 times each. Print out the thread number and current value of count after each increment. Print out the final value after both threads are finished as well. Use POSIX threads and semaphores to implement your solution. Use global variables as appropriate.

**Example**

Output:

Thread 1 incremented count to 1

Thread 2 incremented count to 2

Thread 1 incremented count to 3

Thread 2 incremented count to 4

Thread 1 incremented count to 5

Thread 2 incremented count to 6

Thread 1 incremented count to 7

Thread 2 incremented count to 8

Thread 1 incremented count to 9

Thread 2 incremented count to 10

The value of count is 10

**Solution**

#include <pthread.h>

#include <stdio.h>

#include <semaphore.h>

sem\_t s;

int count;

void \*countup(void \*p) {

int \*id = (int\*)p;

for (int i = 0; i < 5; i++) {

sem\_wait(&s);

count++;

printf(""Thread %d incremented count to %d\n"", \*id, count);

sem\_post(&s);

}

}

int main() {

pthread\_t tid1, tid2;

count = 0;

sem\_init(&s, 0, 1);

int id1 = 1;

int id2 = 2;

pthread\_create(&tid1, NULL, countup, &id1);

pthread\_create(&tid2, NULL, countup, &id2);

pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

printf(""The value of count is %d\n"", count);

}

### Parallelism comparison

Chenyu Xu.

For the following forms of machine parallelism, give an example of an application that will likely work better with this form of parallelism.

1. instruction level parallelism
2. multiplexing
3. process parallelism

#### Solution

1. When we have one CPU and core, and our program does not rely on sequences of values a lot, instruction level parallelism enable micro operations to perform at the same time. This parallelism is enabled besides other machine parallelisms, and is achieved based on techniques of coding.
2. When we have only one CPU and cor, and we need controls over the application. For example, we want to give preferred service or priority to specific client and part of the program. Also, when the whole application need to share data between flows, multiplexing enables every logical flow has access to the entire address space of the process.
3. When we have several CPUs and cores, and have several clients working on the application. Besides enabling parallelism for each client, every client has its seperate address space, so none of them can overwrite other's memory, and keep their memory safe.

### Type Conversion

Contributed by Vince Wu.

Write a function u2f in C that takes in an unsigned and returns a float that has the same bit representation as the unsigned. Hint: use a union.

#### Solution

Unions can be used to read the same bit pattern as different types.

float u2f(unsigned u)

{

union {

unsigned x;

float y;

} convert;

convert.x = u;

return convert.y;

}

### Floating Point Arithmetic

Contributed by James Mullen.

Describe how the computer adds two floating point numbers. Describe how the computer multiplies two floating point numbers.

#### Solution

Addition: Given (-1)^S1 M1 2^E1 + (-1)^S2 M2 2^E2 = (-1)^S M 2^E, assume E1 > E2. Align M1 and M2 based on binary, then add them together. S and M result from the addition. Since E1 > E2, E = E1 initially. If M >= 2, right shift M and increment E. If M < 1, left shift M by k until 1 <= M < 2, then decrement E by k. Overflow if E is out of range and round M to fix fraction precision.

Multiplication: Given (-1)^S1 M1 2^E1 \* (-1)^S2 M2 2^E2 = (-1)^S M 2^E. S = S1 ^ S2, M = M1\*M2, and E = E1 + E2. If M >= 2, right shift M and increment E. If E is out of range, overflow to infinity. Round M to fit fraction precision.

### Processes vs. Threads

Contributed by Eric Tan.

Processes and threads are similar in that they are both sequences of execution. What are the differences between the two, and what are the advantages and disadvantages?

#### Solution

Different processes have different address spaces, while different threads in the same process share the same address space. Thus, communication between threads requires little work since they both have access to shared data structures and variables, while interprocess communication is relatively expensive and requires use of system calls. However, since threads share an address space, an error in one thread can potentially cause errors in the other threads, whereas an error in one process won't affect another independent one. Threads are less expensive to create than processes, so it would be appropriate to use threads when you want multiple computations on some related task, whereas a process might be more appropriate if you want to do tasks that are widely different from each other or require more intensive computation.

**Positive 8-bit floating point values**

*Contributed by William Shao*.

Consider an 8-bit IEEE floating point representation with a leading sign bit, four exponent bits, and three fraction bits.

What is the bias of our 8-bit floating point?

Write the binary representations of the following:

1. Smallest positive number
2. Largest denormalized number
3. Smallest normalized number
4. Decimal value of one
5. Largest normalized number
6. Infinity

**Example**

Write your number with sign, followed by exponent bits, followed by fraction bits. For example, the number 0: 0 0000 000

**Solution**

Bias is 2^(k-1) - 1 where k is the number of exponent bits. So for our 8-bit floating point with four fraction bits, we have a bias of 2^(4-1) - 1 = 8-1 = 7.

1. The smallest positive number will be a denormalized number with just a single bit in the fraction. This gives us: 0 0000 001
2. The largest denormalized number has all zeroes in the exponent and all ones in the fraction: 0 0000 111
3. The smallest normalized number has no fraction and a one in the exponent to make it normalized. 0 0001 000
4. To get a decimal value of one, we should have no fraction and a exponent value that gives us 2^0. Since our bias is 7, we need the exponent bits to hold 7: 0 0111 000
5. The largest normalized number will have a full fraction bit sequence and normalized bit sequence. However, if we fill our normalized bits with all ones we get a special case (infinityor NaN). So the highest exponent bit sequence that remains a normalized number is all ones minus one: 0 1110 111
6. Infinity is represented by all ones in the exponent bits and a zero fraction bit sequence: 0 1111 000