WIKIPEDIA
The Free Encyclopedia

Busy waiting

In computer science and software engineering, **busy-waiting**, **busy-looping** or **spinning** is a technique in which a process repeatedly checks to see if a condition is true, such as whether keyboard input or a lock is available. Spinning can also be used to generate an arbitrary time delay, a technique that was necessary on systems that lacked a method of waiting a specific length of time. Processor speeds vary greatly from computer to computer, especially as some processors are designed to dynamically adjust speed based on current workload.[1] Consequently, spinning as a time-delay technique can produce unpredictable or even inconsistent results on different systems unless code is included to determine the time a processor takes to execute a "do nothing" loop, or the looping code explicitly checks a real-time clock.

In most cases spinning is considered an anti-pattern and should be avoided,[2] as processor time that could be used to execute a different task is instead wasted on useless activity. Spinning can be a valid strategy in certain circumstances, most notably in the implementation of spinlocks within operating systems designed to run on SMP systems.

Example C code

The following C code examples illustrate two threads that share a global integer **i**. The first thread uses busy-waiting to check for a change in the value of **i**:

```
#include <pthread.h>
#include <stdatomic.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* i is global, so it is visible to all functions. It makes use of the special
 * type atomic_int, which allows atomic memory accesses.
 */
atomic_int i = 0;

/* f1 uses a spinlock to wait for i to change from 0. */
static void *f1(void *p)
{
    int local_i;
    /* Atomically load current value of i into local_i and check if that value
       is zero */
    while ((local_i = atomic_load(&i)) == 0) {
        /* do nothing - just keep checking over and over */
    }

    printf("i's value has changed to %d.\n", local_i);
    return NULL;
}

static void *f2(void *p)
{
    int local_i = 99;
    sleep(10);   /* sleep for 10 seconds */
    atomic_store(&i, local_i);
    printf("t2 has changed the value of i to %d.\n", local_i);
    return NULL;
}

int main()
{
    int rc;
    pthread_t t1, t2;

    rc = pthread_create(&t1, NULL, f1, NULL);
    if (rc != 0) {
        fprintf(stderr, "pthread f1 failed\n");
        return EXIT_FAILURE;
    }

    rc = pthread_create(&t2, NULL, f2, NULL);
    if (rc != 0) {
        fprintf(stderr, "pthread f2 failed\n");
        return EXIT_FAILURE;
    }

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    puts("All pthreads finished.");
    return 0;
}
```

In a use case like this, one can consider using C11's condition variables.

Alternatives

Most operating systems and threading libraries provide a variety of system calls that will block the process on an event, such as lock acquisition, timer changes, I/O availability or signals. Using such calls generally produces the simplest, most efficient, fair, and race-free result. A single call checks, informs the scheduler of the event it is waiting for, inserts a memory barrier where applicable, and may perform a requested I/O operation before returning. Other processes can use the CPU while the caller is blocked. The scheduler is given the information needed to implement priority inheritance or other mechanisms to avoid starvation.

Busy-waiting itself can be made much less wasteful by using a delay function (e.g., `sleep()`) found in most operating systems. This puts a thread to sleep for a specified time, during which the thread will waste no CPU time. If the loop is checking something simple then it will spend most of its time asleep and will waste very little CPU time.

In programs that never end (such as operating systems), infinite busy waiting can be implemented by using unconditional jumps as shown by this NASM syntax: `jmp $`. The CPU will unconditionally jump to its own position forever. A busy wait like this can be replaced with:

```
sleep:
hlt
jmp sleep
```

For more information, see HLT (x86 instruction).

Appropriate usage

In low-level programming, busy-waits may actually be desirable. It may not be desirable or practical to implement interrupt-driven processing for every hardware device, particularly those that are seldom accessed. Sometimes it is necessary to write some sort of control data to hardware and then fetch device status resulting from the write operation, status that may not become valid until a number of machine cycles have elapsed following the write. The programmer could call an operating system delay function, but doing so may consume more time than would be expended in spinning for a few clock cycles waiting for the device to return its status.

See also

- Polling (computer science)

- Non-blocking I/O
- Spinlock
- BogoMips
- volatile variable
- Synchronization (computer science)
- Peterson's algorithm

References

1. "Intel Turbo Boost Technology" (https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html).
2. "Why the 'volatile' type class should not be used" (https://www.kernel.org/doc/html/latest/process/volatile-considered-harmful.html). Archived (https://web.archive.org/web/20171004165504/https://www.kernel.org/doc/html/latest/process/volatile-considered-harmful.html) from the original on 2017-10-04. Retrieved 2013-06-10.

External links

- Description (http://www.opengroup.org/onlinepubs/009695399/functions/pthread_spin_lock.html) from The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition
- Article "User-Level Spin Locks - Threads, Processes & IPC (https://web.archive.org/web/20041211235628/http://www.codeproject.com/threads/spinlocks.asp)" by Gert Boddaert
- Austria SpinLock Class Reference (http://austria.sourceforge.net/dox/html/classSpinLock.html)