Multithreading / Synchronization

Threading

- Operating-System-Concepts[1]
  - **Note:** *concurrency* vs *parallelism*
  - programming callenges for multicore/multi-threaded systems:
    - identifying tasks - what can be divided into separate tasks (ideally independent)
    - balance - tasks performing equal work of equal value; if not worth separate task/thread at all?
    - data splitting - data accessed and changes by tasks must be divided too
    - data dependency - synchronization in case of data dependencies of tasks
    - testing debugging
- Linux-Kernel-Development[2]

  > To the Linux kernel, there is no concept of a thread.
  > Linux implements all threads as standard processes
  > The Linux kernel does not provide any special scheduling semantics or
  > data structures to represent threads. Instead, a thread is merely
  > a process that shares certain resources with other processes.
  - syscall `clone()` to create threads
  - *kernel threads* standard processes that exist solely in kernel-space.

Threading - creation and management of multiple units of execution within a single process.
Binary - dormant program resigin on storage
Process - OS abstraction representing the loaded binary, virtual. mem., kernel resources (fds etc)
Thread - unit of execution withing a process: a virtualized processor, a stack, and program state

- Linux-System-Programming[3]
  - Modern OSs have two fundamental (virtualized) abstractions to user-space:
    - virtualized processsor
      - associated with with threads (and not the process)
    - virtual memory
      - associated with process; hence each process has a unique view of memory
      - however all threads in a process *share* this memory
  - Six primary benefits to multithreading:
    - (1) Programming abstraction - dividing work as assigning to units of execution (threads)
    - (2) Parallelism
    - (3) Improving responsiveness
    - (4) Blocking I/O
    - (5) Context switching - cheaper from thread-to-thread (*intraprocess switching*) than process-to-process
    - (6) Memory savings
  - User-level threading
    - *N:1 threading*, also called *user-level threading*
    - a process with N threads maps to a single kernel process (N:1)
    - scheduler etc is a all user-space code
    - not true prallelism
  - Hyper threading
    - *N:M* threading, also known as *hybrid threading*
    - the kernel provides a native thread concept, while user space also implements user threads

[1] https://codex.cs.yale.edu/avi/os-book/
[2] https://www.oreilly.com/library/view/linux-kernel-development/9780768696974/
[3] https://www.oreilly.com/library/view/linux-system-programming/9781449341527/

- Threading patterns, the two core programming patterns are:
  - *thread-per-connection* - unit of work is assigned to one thread, which "runs until completion"
  - *event-driven*:
    - comes from observation that threads do a lot of waiting: reading files, waiting for dbs, etc
    - suggested to consider the *event-driven pattern* first i.e.
    - asynchronous I/O, callbacks, an event loop, a small thread pool with just a thread per processor.
- *Race condition* situation in which unsynchronized access of shared resources lead to errors
  - shared resources examples: system's hw, a kernel resource, data in memory etc
  - *critical region* - region of code that should be synchronized
- **Synchronization**
  - The fundamental source of races is that critical regions are a window during which correct program behavior requires that threads do not interleave execution. To prevent race conditions, then, the programmer needs to synchronize access to that window, ensuring mutually exclusive access to the critical region.
  - *atomic operation* - indivisible, unable to be interleaved with other operations
    - problem with critical regions: they are not atomic
- **Locks / Mutexes**
  - *lock* mechanism for ensuring mutual exclusion within a cricial region, making it *atomic*
  - also knows as *mutexes* (e.g. in Pthreads)
  - the smaller the critical region, the better:
    - locks prevent concurreny thus negating threading benefits
- ***Deadlock*** - situation where 2 threads are waiting for the other to finish, thus neither does

Synchronization

Operating-System-Concepts[4]

- **race condition**
  - several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place
  - examples in *kernel code*
    - data stracture maintaining a list of all open files
    - variable *next_available_pid*
- **mutex lock** - simplest 'tool' to protect critical sections and prevent race conditions
  - short for **mu**tual **ex**clusion
  - example, *note* calls to `acquire` and `release` must be performed atomacially e.g. using CAS (compare-and-swap)

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

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

  - mutex **lock** states:
    - *contended* thread blocks while trying to aquire lock
    - *uncontended* lock is available when thread attempts to require it

[4] https://codex.cs.yale.edu/avi/os-book/

  - another name for this *type of mutex lock*: **spinlock**
  - can be held by at most one thread of execution
  - main **disadvantage**: ***busy waiting*** (spinlock)
    * process *spins* while waiting for lock (busy wait)
    * advantage: no context switch required:
      · *a context switch to move the thread to the waiting state and a*
      · *second context switch to restore the waiting thread once the lock becomes available*
- **semaphore** - an `int` accessed only through two standard atomic operations: `wait()` and `signal()`
  - from Linux-Kernel-Development[5]:
  - in Linux, sepaphores are *sleeping locks*
    * *When a task attempts to acquire a semaphore that is unavailable, the semaphore places*
    * *the task onto a* ***wait queue*** *and puts the task to sleep.*
  - *pro* because task sleeps, semaphores are well suited to locks that are held for a long time
  - *con* because of overhead (two context switches) not optimal for locks held for short periods
  - often 'used' as sleeping *mutual exclusion lock* ("sleeping spin lock") with a *count* of one
  - *counting semaphore* (value between 0 and 1) - can be used instead of mutex locks
  - can have arbitrary number of simultaneous lock holders
    * spinlock as well as `mutex_lock` at most one task/thread
  - both for critical sections (mutual exclusion) and coordination (scheduling)
- **monitor**
  - provides mutual exclusion; programming language construct (not available in all languages)
  - *"The monitor construct ensures that only one process at a time is active within the monitor."*
  - A monitor is an abstract data type that provides a high-level form of process synchronization. A monitor uses condition variables that allow processes to wait for certain conditions to become true and to signal one another when conditions have been set to true.
  - Examples:
    * in Java part of language; `synchronized` keyword
    * in Rust not part of language but provided via `Struct shared_mutex::monitor::Monitor`
- **liveness**
  - refers to a set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle.
- **deadlock**
  - a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.
  - **prevention** - few simple rules:
    * Implement *lock ordering*; document lock ordering. Always aquire lock in same order (ABAB vs ABBA)
    * Prevent *starvation*. Will code always finish?(or dependent on certain event on potentially wit forever)
    * Do *not double aquire the same lock*
    * *Simplicity* - design for simplicity; complexity in locking schemes 'invites' deadlocks
- locking *approaches*
  - *optimistic* - first update variable and then use collision detection
  - *pessimistic* - assume another thread is concurrently updating the variable...
    * so you pessimistically acquire the lock before making any updates.

Semaphores vs Mutexes vs Spinlock

- Mutexes and Smaphores are simalar; both sleep wait processes
  - use new mutex type whenever possible over semaphore (b/c it's simpler)
- Spinlocks if short lock hold time or in interrupt handler (semaphores can't as they'd sleep)

[5] https://www.oreilly.com/library/view/linux-kernel-development/9780768696974/

Synchronization Examples

Note:

> Prior kernel 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted—even if a higher-priority process became available to run.

- **Kernel**
  - Linux-Kernel-Development[6] *"mutex" is a generic name to refer to any sleeping lock that enforces mutual exclusion*
  - *Atomics*: Linux kernel atomic integer `atomic_t` with atomic operations `atomic_inc(&i)` or `atomic_read(&i)`
  - *Semaphores*:
    - Linux-Kernel-Development[7] "Until recently, the only sleeping lock in the kernel was the semaphore"

      ```
      struct semaphore name;
      sema_init(&name, count);
      ```
  - *Mutex locks*: `mutex_lock()` / `mutex_unlock()`
    - "If the mutex lock is unavailable, a task calling `mutex lock()` is put into a sleep state and is awakened when the lock's owner invokes `mutex unlock()`"
    - Linux-Kernel-Development[8] - behaves similar to a semaphore with a count of one, but it has a simpler interface
  - *Spinlock*

```
<linux/spinlock.h>
spin_lock(S);
/* critical section */
spin_unlock(S);
```

- POSIX **Pthreads**
  - mutex locks data type `pthread_mutex_t`

```
#include <pthread.h>
int main() {
    pthread_mutex_t mutex;
    pthread_mutex_init(&mutex,NULL); /* create + init the mutex lock */
    pthread_mutex_lock(&mutex); /* acquire lock */
    /* ****************************
     * critical section goes here
     * ****************************/
    pthread_mutex_unlock(&mutex); /* release lock */
}
```

- Pthreads deadlock example:

```
$ ps uaxH  # process state: S
archlin+     902  0.0  0.0  18524   568 pts/0    Sl+  10:28   0:00 ./a.out
# PROCESS STATE CODES
# S    interruptible sleep (waiting for an event to complete)
# l    is multi-threaded
```

```
#include <pthread.h>
```

[6] https://www.oreilly.com/library/view/linux-kernel-development/9780768696974/
[7] https://www.oreilly.com/library/view/linux-kernel-development/9780768696974/
[8] https://www.oreilly.com/library/view/linux-kernel-development/9780768696974/

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

pthread_mutex_t mutex_A;
pthread_mutex_t mutex_B;

void *thread_function_01(void *param) {
    pthread_mutex_lock(&mutex_A);
    sleep(1); /* this will force deadlock as thread02 will lock mutex_B  */
    pthread_mutex_lock(&mutex_B);
        printf ("%s\n", "thread01"); /* do work */
    pthread_mutex_unlock(&mutex_B);
    pthread_mutex_unlock(&mutex_A);
    pthread_exit(0);
}
void *thread_function_02(void *param) {
    pthread_mutex_lock(&mutex_B);
    pthread_mutex_lock(&mutex_A);
        printf ("%s\n", "thread02"); /* do work */
    pthread_mutex_unlock(&mutex_A);
    pthread_mutex_unlock(&mutex_B);
    pthread_exit(0);
}
int main(void) {
    pthread_t thread_01, thread_02;
    pthread_mutex_init(&mutex_A, NULL);
    pthread_mutex_init(&mutex_B, NULL);
    pthread_create(&thread_01, NULL, thread_function_01, (void *) NULL);
    pthread_create(&thread_02, NULL, thread_function_02, (void *) NULL);
    pthread_join (thread_01, NULL);
    pthread_join (thread_02, NULL);
    return 0;
}
```

- **livelock**
  - a liveness failure
  - *livelock occurs when a thread continuously attempts an action that fails.*
  - *typically occurs when threads retry failing operations at the same time.*
  - *can generally be avoided by having each thread retry the failing operation at random times*
  - from Java documentation: > A thread often acts in response to the action of another thread. > If the other thread's action is also a response to the action of another thread, > then livelock may result.