Systems 3

Threads

Marcel Waldvogel

(Handout)

Department of Computer and Information Science
University of Konstanz

Winter 2019/2020

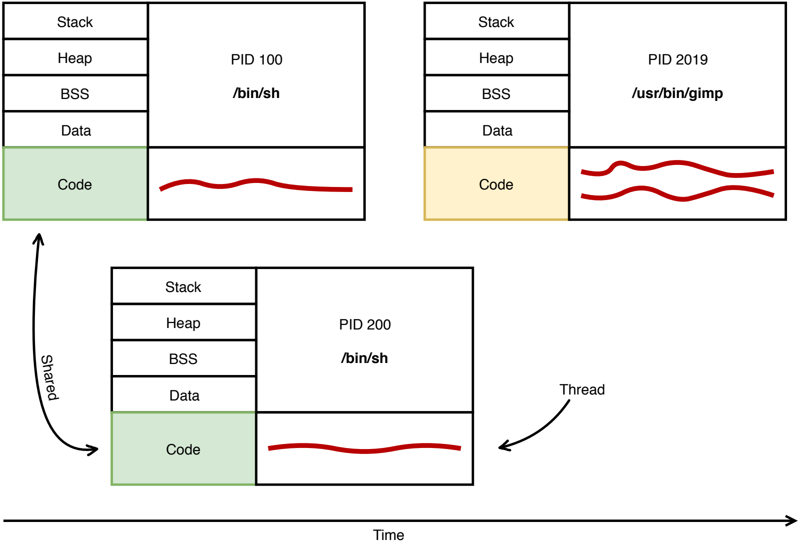
These slides are based on previous lectures held by Alexander Holupirek, Roman Byshko, and especially Stefan Klinger.

Chapter Goals

- The power of threads
- The dangers and pitfalls of threads
- How to use threads

- Mutex, Read-Write Locks, Condition Variable, and Semaphore:
  - How to use them
  - How they compare/differ

Revisited: Threads vs. Processes vs. Programs



Pthreads API

```c
#include <pthread.h>
pthread_create(pthread_t *thread,
    const pthread_attr_t *attr, void *(start)(void *), void *arg);
```

- POSIX defines an API for threads in `<pthread.h>`. (use `gcc -lpthread` to compile.)
- `pthread_create` creates a new thread that runs the function `start` with the provided argument(s) in `arg`.
- Thread will run independently in parallel, unless it decides it needs synchronization.
- A unique thread identifier will be written in the `pthread_t` buffer provided.
- Attributes are specified in `attr` and `NULL` is the default.
- `arg` is usually the address of a `struct` that is then typecast back inside the `start` function, to allow for multiple arguments.

```
void pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **retval);
int pthread_detach(pthread_t thread );
pthread_t pthread_self(void);
int pthread_equal(pthread_t t1, pthread_t t2);
```

- Threads can exit by returning from the `start` function or calling `pthread_exit()`.
- Its return value can be retrieved with `pthread_join()`. This **blocks** if the thread is still running.
- If the thread's return value is irrelevant, it can use `pthread_detach()`. It will then be directly cleaned up after exiting, as to not hog resources. (cf. zombies)
- A thread's own identifier can be retrieved using `pthread_self()`.
- Two thread identifiers can be compared with `pthread_equal()`.

```
static int global = 0;
static void *start(void *arg) {
  int loops = *((int *) arg);
  for (int j = 0; j < loops; j++) {
    local = global; // Even "global++" would not be atomic
    local++;
    global = local;
  }
  return NULL;
}

int main(void) {
  pthread_t t1, t2;
  int loops = 1000;
  if (pthread_create(&t1, NULL, start, &loops) != 0) err(1, "create 1");
  if (pthread_create(&t2, NULL, start, &loops) != 0) err(1, "create 2");
  if (pthread_join(t1, NULL) != 0) err(1, "join 1");
  if (pthread_join(t2, NULL) != 0) err(1, "join 2");
  printf("global = %d", global);
  exit(EXIT_SUCCESS);
}
```

Results on this machine

Run	local++	global++
1	2000	2000
2	2000	2000
3	1385	2000
4	1401	2000
5	1645	2000
6	1846	2000
7	1761	2000
8	1956	2000
9	1665	2000
10	2000	1958
min	1385	1958
max	2000	2000
avg	1766	1996

Which is more dangerous?

Even if it works for you today, it may not work for anyone else or for you tomorrow.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <err.h>

#define LOOPS 1000

static int global = 0;

struct args {
   int loops;
   int thread_no;
} a0={LOOPS, 0}, a1={LOOPS, 1};

static void *start(void *arg) {
   struct args *a=arg;
   for (int j = 0; j < a->loops; j++) {
      while ((global & 1) == a->thread_no) {
      }
      global++;
   }
   return NULL;
}
...
```

Thread-shared variables

Optimizer

The compiler's optimizer tries to remove unnecessary instructions and memory accesses.

`for (int i=0; i<N; i++) x++;` → `x += N;`

Memory accesses can be forced with `volatile`.

When to `volatile` a variable?

- When a variable is shared between multiple threads
- ...with an interrupt handler
- ...with a signal handler
- ...with hardware

Synchronisation: Mutex

```c
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_lock(pthread_mutex_t * mutex);
int pthread_mutex_unlock(pthread_mutex_t * mutex);
```

- Mutexes provide a simple locking mechanism for some resource.
- Mutexes must be initialized; Either statically with `PTHREAD_MUTEX_INITIALIZER` or dynamically calling `pthread_mutex_init()`.
- The critical region is between `pthread_mutex_lock()` and `pthread_mutex_unlock()`.
- `pthread_mutex_lock()` will block if the mutex is currently locked and return once it is available.
- `pthread_mutex_unlock()` releases the lock respectively.

```c
int pthread_mutex_trylock(pthread_mutex_t * mutex);
int pthread_mutex_timedlock(pthread_mutex_t * mutex,
const struct timespec *abs_timeout);
```

- `pthread_mutex_trylock()` will not block if the mutex is locked, but return `EBUSY`
- `pthread_mutex_timedlock()` will return when the lock is aquired, or at least the time specified in `abs_timeout` has passed, in which case `ETIMEDOUT` is returned.
- Both are rare and if you think they suit your needs, you should at least consider the other options presented in this chapter and confirm that your critical regions are indeed minimal.
- `trylock` and `timedlock` variants are also available for other synchronisation methods in this chapter, but will not be introduced as they behave analogically.

Behaviour:

- A locked mutex must not be locked again before unlocking.
- A thread must not unlock a mutex it doesn't own the lock to.
- An unlocked mutex must not be unlocked.
- `PTHREAD_MUTEX_INITIALIZER` must only be used for static mutexs i.e. globally.
- Dynamic mutexs must be initialized and destroyed with `pthread_mutex_init()` and `pthread_mutex_destroy()` respectively.

```c
static int global = 0;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

static void *start(void *arg) {
  int loops = *((int *) arg);
  int local;
  for (int j = 0; j < loops; j++) {
    if (pthread_mutex_lock(&mutex) != 0)
      exit(EXIT_FAILURE);
    local = global;
    local++;
    global = local;
    if (pthread_mutex_unlock(&mutex) != 0)
      exit(EXIT_FAILURE);
  }
  return NULL;
}
```

Synchronisation: RW-Locks

- Locking prevents parallelization, thus slows down performance.
- What about a state that is rarely changed but often read?
- Consider a database that contains products in a webshop:
  - $\Rightarrow$ One read for every access, one write for every purchase.
  - $\Longrightarrow$ Allow unlimited reads in parallel, but only one write, during which no read must occur either.

```
pthread_rwlock_t lock_rw = PTHREAD_RWLOCK_INITIALIZER;
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Behaviour: Similar to mutex with some exceptions:

- A thread can hold multiple readlocks.
- `pthread_rwlock_unlock()` must be called for each readlock to unlock.

Implementations have to prevent **Writer's Starvation**, i.e., writers never getting a lock because new readers keep coming.

How?

Synchronisation: Condition Variable

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

- Canonical notion of *waking up* another thread (Producer-Consumer) is complicated to implement with locks.
  $\Longrightarrow$ **Condition Variable** (**Monitor** in literature)
- `pthread_cond_wait()` requires the thread to hold a lock on `mutex`.
- It will then release the lock and block atomically.
- A waiting thread is woken up by a thread calling `pthread_cond_signal()`
- If multiple threads are waiting, there is no guarantee which **one** will receive the signal or that only **one** is woken up → Recheck (`while`)!
- `pthread_cond_broadcast()` wakes up all waiting threads.

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static int avail = 0;

void *produce(void *arg) {
  for (int i = 0; i < *(int *)arg; i++) {
    sleep(1); //produce
    if (pthread_mutex_lock(&mtx) != 0) exit(EXIT_FAILURE);
    avail++;
    if (pthread_mutex_unlock(&mtx) != 0) exit(EXIT_FAILURE);
    // Nota bene
    if (pthread_cond_signal(&cond) != 0) exit(EXIT_FAILURE);
  }
  return NULL;
}

void *consume(void *arg) {
  for (;;) {
    if (pthread_mutex_lock(&mtx) != 0) exit(EXIT_FAILURE);
    while (avail == 0) { // Test always done with &mtx locked!
      if (pthread_cond_wait(&mtx, &cond) != 0) exit(EXIT_FAILURE);
    }
    /* we have a lock on mtx */
    while (avail > 0) {
      // consume here, but let's not call sleep, because we have a lock
      avail--;
    }
    if (pthread_mutex_unlock(&mtx) != 0) exit(EXIT_FAILURE);
  }
  return NULL;
}
```

Synchronisation: Semaphores

Definition A semaphore is a *shared integer* variable that *cannot drop below zero*.

- It is **always** possible to **increment** a semaphore (within the integer bounds).
- If the semaphore is **zero**, a **decrement blocks** until another process increments the semaphore.

Iterfaces Linux provides different semaphore interfaces.

- See `sem_overview`(7) for an overview.
- We discuss only one form:

Named POSIX semaphores

- Identified by a name, with exactly one leading slash, *e.g.*, `/name`.
- Linux stores them in the file system at `/dev/shm/sem.name`.

```c
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

- `sem_open`(3) opens, creates, and initializes a named semaphore:
  - `oflag` is ored together with constants from `fcntl.h`:
    - `O_CREAT` create semaphore iff it does not exist.
    - `O_EXCL` fail if `O_CREAT` and semaphore exists.
  - Iff the semaphore is created, it's initialized to `value`, and the permissions are set to `mode & umask` (see `sys/stat.h`).
- `sem_close`(3) closes a previously opened semaphore.
- `sem_unlink`(3) removes a named semaphore from the system.
  - It cannot be opened by other programs any more.

Note Named POSIX semaphores are **kernel persistent**, *i.e.*, if not removed with `sem_unlink`(3), they live until reboot!

```c
#include <semaphore.h>
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
```

- sem_post(3) **increments** the semaphore pointed to by sem.
- sem_wait(3) **decrements** the semaphore pointed to by sem.
  - This may block until sem is greater than zero to allow for the decrement.
  - Nonblocking functions sem_trywait(3) and sem_timedwait(3) are available.
- All return 0 on success. On error, −1 returned, and errno is set.

Example

```c
int main(void)
{
    sem_t *s = sem_open("/semaphore", O_CREAT, S_IRUSR|S_IWUSR, 3);
    if (s == SEM_FAILED)
        err(1, "sem_open");

    printf("semaphore[%d]: waiting\n", getpid());
    if (sem_wait(s) < 0)
        err(1, "sem_wait");

    printf("semaphore[%d]: critical section\n", getpid());
    sleep(15);  /* this is the critical section */

    printf("semaphore[%d]: posting\n", getpid());
    if (sem_post(s) < 0)
        err(1, "sem_post");

    if (sem_close(s) < 0)
        err(1, "sem_close");

    return 0;
}
```

```
$ gcc -o semaphore -lpthread semaphore.c        # links with required library
$ ./semaphore
semaphore[4056]: waiting
semaphore[4056]: critical section
semaphore[4056]: posting
```

- Try this in **multiple terminals** in parallel. Only three processes will be in the “critical section” at the same time.
- Do not forget to **remove the semaphore** after using!
  - Either on the command line: `rm /dev/shm/sem.semaphore` (OS dependent!),
  - Or use the C interface (portable):

```
int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++)
        if (sem_unlink(argv[i]) != 0)
            err(1, "sem_unlink(%s)", argv[i]);
    return 0;
}
```

```
$ gcc -o rm_sem -lpthread rm_sem.c
$ ./rm_sem /semaphore
```