Programmation Système
Cours 9 — POSIX Threads

an introduction

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2014–2015

URL http://upsilon.cc/zack/teaching/1415/progsyst/
Copyright © 2012–2015 Stefano Zacchiroli
License Creative Commons Attribution-ShareAlike 4.0 International License
http://creativecommons.org/licenses/by-sa/4.0/deed.en_US

Outline

1. Threads and Pthreads
2. Mutex
3. Condition variables
4. Thread safety
5. Going further

Outline

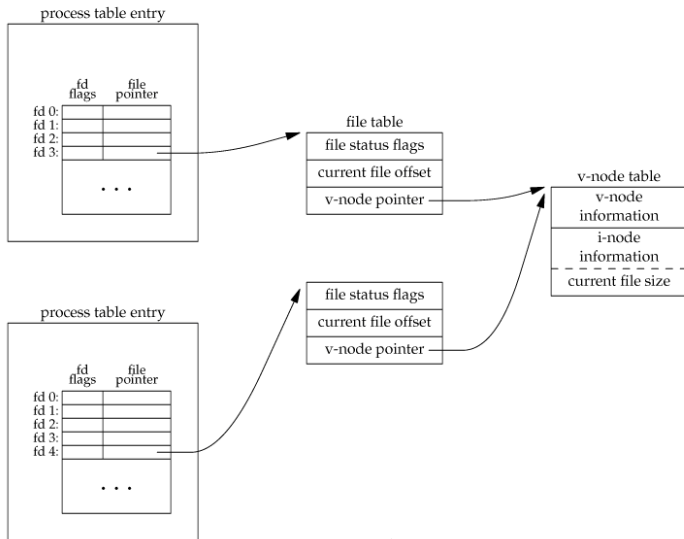
1. Threads and Pthreads
2. Mutex
3. Condition variables
4. Thread safety
5. Going further

Looking back

We have already seen the notion of thread,
at the very beginning of this course. . .

Redux: multi-process I/O — file sharing

What happens when *independent* processes act on the same file?



APUE, Figure 3.7

Redux: sharing resources among processes

It seems processes can share resources, such as the v-node table in the previous (degenerate) example.

Can they share more?

Redux: sharing resources among processes

It seems processes can share resources, such as the v-node table in the previous (degenerate) example.
Can they share more?
Yes.

- related processes can share file table entries
- processes can share specific memory regions of their own address space (e.g. memory mapping, shared memory)
- pushing it to the extreme, multiple "processes" can share by default their entire address space by the means of threads

Redux: sharing resources among processes

It seems processes can share resources, such as the v-node table in the previous (degenerate) example.
Can they share more?
Yes.

- related processes can share file table entries

we now have seen how to do this (e.g. via `fork` inheritance)...

- processes can share specific memory regions of their own address space (e.g. memory mapping, shared memory)

...and this (e.g. via `mmap` or `shmat`...)

- pushing it to the extreme, multiple "processes" can share by default their entire address space by the means of threads

...how about this?

Redux: threads

By default, each process has only one thread of control (or “thread”), i.e. only one set of instructions being executed at any given time.

Additional threads can be added at runtime.

Threads and processes

Like processes, threads:

- are a mechanism to allow applications to perform concurrent tasks...
  - e.g. doing CPU-intensive calculations...while updating the user interface
- possibly in a *truly concurrent* manner (as opposed to time sharing) on a multiprocessor system
  - e.g. doing CPU-intensive calculations...while waiting for I/O

Unlike processes:

- threads are all executed in the same address space
  - whereas processes had one address space each, copy-on-write'd from the parent

Intuition

- new process = new execution point + new address space
- new thread = new execution point

Why bother?

We have seen that, thanks to copy-on-write, `fork` is very fast.
Why bother with threads?

- sharing information requires work — that's why we have spent one semester studying IPC mechanisms...
- we can be faster than `fork` — even with COW, the kernel has to duplicate the page table (and more)

Why bother?

We have seen that, thanks to copy-on-write, fork is very fast.
Why bother with threads?

- sharing information requires work — that's why we have spent one semester studying IPC mechanisms...
- we can be faster than fork — even with COW, the kernel has to duplicate the page table (and more)

With threads:

- thread creation is much faster than process creation, $\approx$10x:[1]

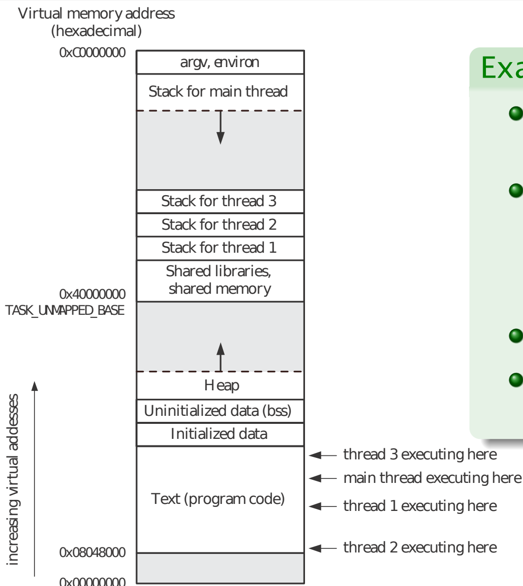
100'000 fork	22.27s
100'000 vfork	3.52s
100'000 "thread creations"	2.97s

- we have sharing by default — global or heap variables are accessible by all threads

drawback: we will need synchronization—badly and heavily—to avoid race conditions and memory corruption

[1] on Linux 2.6.27, x86-32. Source: TLPI §28.3 (2010)

Memory layout (approx.)



TLPI, Figure 29-1

Example

- 4 threads in execution
  - the main one + 3 extra
- . . . with 4 independent execution points
  - i.e. 4 independent program counter (PC) values
- . . . with 4 independent stacks
- . . . all sharing the same address space

Sharing. . .

Some shared traits, among threads of the same process:

- address space
- pid, parent pid, and most user info
  - e.g. credentials, group, session, . . .
- open file descriptors
- most file system info
  - e.g. current working directory, file locks, . . .
- signal dispositions

. . . or not sharing

Some non-shared traits, private to each thread of the same process:

- thread ID
- stack
  - note: this means local variables are *not* shared either
  - no encapsulation enforcement, though; beware of pointers (black) magic
- `errno`
- signal mask
- thread-local storage

Pthreads API

Pthreads = POSIX thread API

In the 80–90s various APIs existed to do thread programming on UNIX. POSIX.1c (1995) blessed Pthreads ("POSIX threads") as the standard thread API on UNIX. Later on, Pthreads have also been integrated into SUSv3.

Main parts of Pthreads:

- abstract data types (`pthread_t`, `pthread_mutex_t`, ...)
- thread management (creation, "wait", cancellation, ...)
- mutex
- condition variables
- much more (read/write locks, memory barriers, ...)

very large and complex API! we will only scratch the surface...

errno and threads

1. errno is a global integer variable
2. errno depends on function return values
3. threads = independent execution points + same address space

errno and threads

1. errno is a global integer variable
2. errno depends on function return values
3. threads = independent execution points + same address space

FAIL.

errno and threads

1. errno is a global integer variable
2. errno depends on function return values
3. threads = independent execution points + same address space

FAIL. or not...

Gory details

- in threaded programs ernno is actually a macro
- ...that expands to a function call
- ...which returns a modifiable lvalue

Bottom line

errno has been adapted to cope well with threads: each thread has its own errno. Don't worry about it.

Return value convention

Usual convention

- success: return 0 (or some positive informative value)
- failure: return -1; set `errno`

Pthread API convention

- success: return 0
- failure: return `errno` value > 0

Why? Because.
(and also because every `errno` access in threaded programs costs you a function call overhead)

Thread-specific helper functions

```
void err_msg_p(int errnum, const char *msg) {
        char buf[BUFSIZ];

        if (strerror_r(errnum, buf, BUFSIZ) != 0)
                fprintf(stderr, "%s: Unknown error %d\n",
                        msg, errnum);
        else
                fprintf(stderr, "%s: %s\n", msg, buf);
}

void err_sys_p(int errnum, const char *msg) {
        err_msg_p(errnum, msg);
        exit(EXIT_FAILURE);
}
```

Sample usage:

```
pthread_t thread;
int s;
s = pthread_func(&thread, ...);
if (s != 0)
    err_sys_p(s, "pthread_func");
```

Compiling with Pthreads (on Linux)

On Linux, programs using Pthreads must be compiled passing the `-pthread` option to the C compiler:

> **-pthread**
> *Adds support for multithreading with the pthreads library. This option sets flags for both the preprocessor and linker.*
>
> — *manpage CC(1)*

Why?

Intuition: you need to tell the compiler toolchain to do all sort of thread-specific treatments, like the `errno` trick.

Thread creation

The main thread exists by default. We can add extra threads with:

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
             void *(*start)(void *), void *arg);
```

Returns: *0 on success, positive errno on error*

What does the function prototype tell us?

Thread creation

The main thread exists by default. We can add extra threads with:

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start)(void *), void *arg);
```

Returns: *0 on success, positive errno on error*

- *start* points to the thread code (i.e. a function)
- *arg* will be passed as argument to *start*
  - the new thread will start executing `start(arg)`
  - the calling thread will continue execution after `pthread_create`
  - the usual "who goes first?" race condition applies
- *thread* will be filled with a thread ID identifying the just created thread, before `pthread_create` returns
- *attr* can be used to specify new thread attributes [. . .]

Thread termination

A thread terminates its execution when:

- its "main" function terminates (using `return` or implicitly)
- the thread is canceled using `pthread_cancel(...)`
- *any* of the thread calls `exit` → *all* threads will terminate (!)
- the thread calls `pthread_exit`:

Thread termination

A thread terminates its execution when:

- its "main" function terminates (using `return` or implicitly)
- the thread is canceled using `pthread_cancel (...)`
- *any* of the thread calls `exit` → *all* threads will terminate (!)
- the thread calls `pthread_exit`:

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Returns: *does not return*

- *retval* will be available for inspection by other threads
- note: the thread creation/termination API is more expressive than `fork/exit` (why?)

Thread “waiting”

After the thread-equivalent of `fork`, let’s see `waitpid`’s:

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

Returns: *0 on success, positive errno on error*

- calling thread will block until *thread* terminates
- if not NULL, *retval* will get a copy of *thread* return value

Notable differences with respect to `wait(pid)`

- threads are peers not parents/children: anyone can join anyone
- there is no direct equivalent of `wait`, to join with any thread

Pthreads — example

```c
#include <pthread.h>
#include "helpers.h"
void *tmain(void *arg) {
        char *msg = (char *) arg;
        printf("[thread] %s", msg);
        return ((void *) strlen(msg));
}
int main(int argc, char **argv) {
        pthread_t t;
        int s;
        void *res;

        printf("creating thread...\n");
        s = pthread_create(&t, NULL,
                           tmain, (void *) "Hello, world!\n");
        if (s != 0) err_sys_p(s, "pthread_create");
        s = pthread_join(t, &res);
        if (s != 0) err_sys_p(s, "pthread_join");
        printf("thread returned %ld\n", (long) res);
        exit(EXIT_SUCCESS);
} /* end of hello_pthread.c */
```

Demo

- argument passing between threads (forth and back)
- judicious casting to return integers (still dangerous, though. . .)

Pthreads — larger example

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

void *tmain(void *arg) {
        long id = (long) arg;

        printf("[thread %ld] Hello, World!\n", id);
        return ((void *) id);
}

int main(int argc, char **argv) {
        int s;
        long i, threads_no;
        pthread_t *ts = NULL;   // pthread_t array

        if (argc != 2)
                err_quit("Usage: hellos_pthread THREAD_NO");
        threads_no = atol(argv[1]);      // TODO: error checking

        ts = calloc(threads_no, sizeof(pthread_t));
        if (ts == NULL)  err_sys("malloc");
```

Pthreads — larger example (cont.)

```
        for (i = 0; i < threads_no; i++) {
                printf("creating thread %ld...\n", i);
                s = pthread_create(ts + i, NULL,
                                   tmain, (void *) i);
                if (s != 0) err_msg_p(s, "pthread_create");
        }

        for (i = 0; i < threads_no; i++) {
                void *res;
                s = pthread_join(ts[i], &res);
                if (s != 0) err_msg_p(s, "pthread_join");
                printf("thread %ld returned %ld\n", i, (long) res);
        }

        free(ts);
        exit(EXIT_SUCCESS);
} /* end of hellos_pthread.c */
```

Demo

- starts *all* threads before joining, to maximize parallelism
- needs explicit thread (id) management
- execution is non deterministic, as expected

Zombie threads

Definition (thread zombie)

A thread is said to be a zombie thread if it has terminated and no other thread has collected its return value (with `pthread_join`) yet.

Zombie threads pose very similar problems to zombie processes—waste of system resources and, eventually, thread ID exhaustion (making impossible to create new threads).

Detaching threads

To avoid thread zombies:

- use `pthread_join` judiciously, or
- make the thread non-joinable using:

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Returns: *0 on success, positive* `errno` *on error*

Non-joinable threads are automatically reaped at termination (but it is no longer possible to collect their return values).

Note: at `exit`, joinable and non-joinable threads are terminated all alike.

Thread identity

When doing thread management, it might come handy to answer the question "Who am I ?"

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Returns: *thread ID of the calling thread*

```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

Returns: *nonzero value if t1 and t2 are equal, 0 otherwise*

Note: `pthread_t` is an abstract data type, with different implementations on different UNIX. You can't use == on `pthread_t` values, and there is no portable way of printing them.

- you need a printable thread id?
  pass one to your threads as we did in the example!

Outline

1. Threads and Pthreads
2. Mutex
3. Condition variables
4. Thread safety
5. Going further

Threads and critical sections

Reminder:

Definition (critical section)

A critical section is a code snippet that accesses a shared resource and that shall be executed atomically with respect to other (kernel scheduling) “entities” (KSE) accessing the same shared resource.

Up to now:

- entities = processes
- shared resources = those sharable between processes
  - specific memory pages, file descriptors, SysV structures, etc.

Now we *add*:

- entities = threads
- shared resources = all non-local variables by default (!)

Threads and critical sections — example

Let's consider a simple example:

- a process address space contains the global variable
  static long glob = 0;
- 2 threads of the same process wants to increment the global variable in parallel, as follows:

```
void *tmain(void *arg) {
        long loc;

        loc = glob;
        loc++;
        glob = loc;
}
```

Threads and critical sections — example

Let's consider a simple example:

- a process address space contains the global variable
  static long glob = 0;
- 2 threads of the same process wants to increment the global variable in parallel, as follows:

```
void *tmain(void *arg) {
        long loc;

        loc = glob;
        loc++;
        glob = loc;
}
```

What could possibly go wrong?

Threads and critical sections — example

Let's consider a simple example:

- a process address space contains the global variable
  static long glob = 0;
- 2 threads of the same process wants to increment the global variable in parallel, as follows:

```
void *tmain(void *arg) {
        long loc;

        loc = glob;
        loc++;
        glob = loc;
}
```

What could possibly go wrong?

To experimentally observe the problem, we will use 2 threads and iterate the variable update code an increasing number of times...

Threads and critical sections — example (cont.)

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

static long glob = 0;

void *tmain(void *arg) {
        long n = (long) arg;
        long i, loc;

        for (i = 0; i < n; i++) {
                loc = glob;
                loc++;
                glob = loc;
        }

        return NULL;
}
```

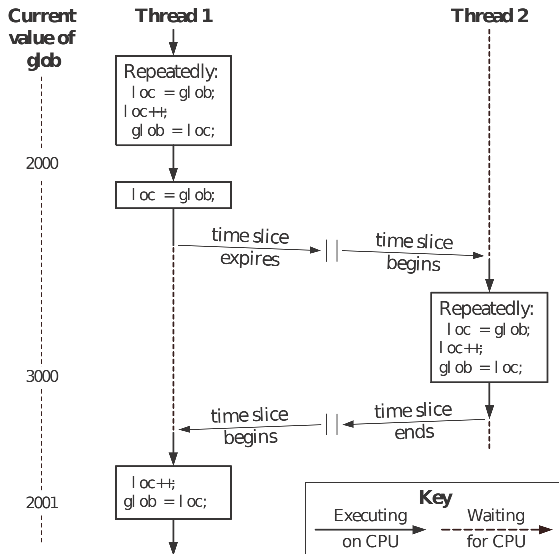
Threads and critical sections — example (cont.)

```c
int main(int argc, char **argv) {
	pthread_t t1, t2;
	int s;
	long n;
	if (argc != 2)
		err_quit("Usage: pthread_race ITERATIONS");
	n = atol(argv[1]);	// TODO: error checking

	s = pthread_create(&t1, NULL, tmain, (void *) n);
	if (s != 0) err_sys_p(s, "pthread_create (t1)");
	s = pthread_create(&t2, NULL, tmain, (void *) n);
	if (s != 0) err_sys_p(s, "pthread_create (t2)");

	s = pthread_join(t1, NULL);
	if (s != 0) err_sys_p(s, "pthread_join (t1)");
	s = pthread_join(t2, NULL);
	if (s != 0) err_sys_p(s, "pthread_join (t2)");

	printf("glob = %ld (expected: %ld)\n", glob, n * 2);
	exit(EXIT_SUCCESS);
} /* end of pthread_race.c */
```

Threads and critical sections — example (cont.)

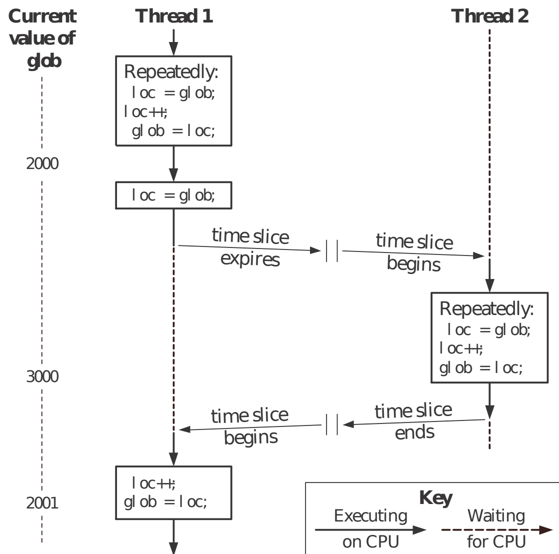
Demo

Threads and critical sections — example (cont.)



TLPI, Figure 30-1

Threads and critical sections — example (cont.)

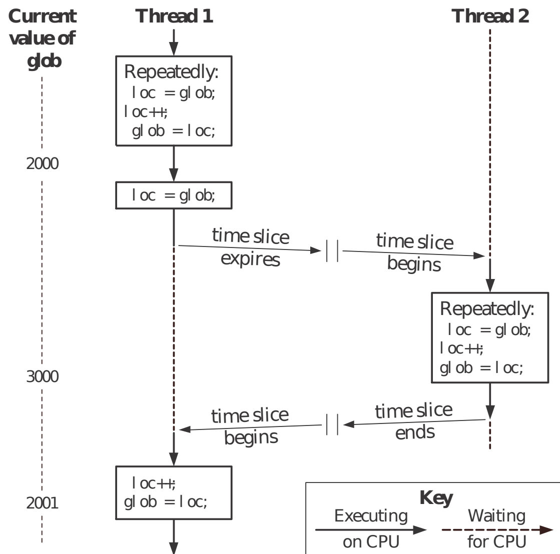


TLPI, Figure 30-1

Exercise

- *Why does it work properly for a small enough number of iterations?*

Threads and critical sections — example (cont.)



TLPI, Figure 30-1

Exercise

- *Why does it work properly for a small enough number of iterations?*

Exercise

- *Would replacing the 3-instruction update with `glob++` solve the problem?*

Thread mutex

As usual, to *enforce* critical sections, we need some mutual exclusion mechanism.

The Pthreads API offers mutex as a fast synchronization mechanism to enforce mutual exclusion between threads.

Terminology and “protocol”:

1. threads lock mutex
   - AKA: “acquire”
2. critical section
3. threads unlock mutex
   - AKA: “release”

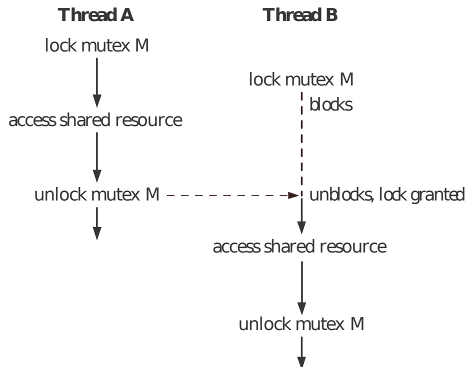


Figure: sample interaction between 2 threads trying to access the same mutex (and shared resource)

Creating mutex

Pthread mutex creation depends on whether you want to allocate them statically (i.e. in a global variable) or dynamically (i.e. in a `malloc`-ed memory area).

Static mutex allocation is easier, just declare a global variable:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

Drawback: you will get a mutex initialized with various default mutex attributes.

Creating mutex (cont.)

Dynamic mutex allocation is more flexible and is achieved using:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
             const pthread_mutexattr_t *restrict attr);
```

Returns: *0 on success, positive errno on error*

- where *mutex* points to dynamic memory (i.e. either allocated on the heap with `malloc` or a local variable allocated on the stack)
- *attr* (when non NULL) allows to specify desired non-default mutex attributes

Destroying mutex

Dynamically allocated mutex should be destroyed when no longer needed using:

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Returns: *0 on success, positive errno on error*

For your health:

- destroy a mutex only when it is unlocked, and
- no other thread will lock it further in the future

Mutex types

The most relevant mutex attribute is the mutex type. On it depend the results of the following corner case operations:

1. trying to lock a mutex already locked by yourself
2. trying to unlock a mutex that is not locked
3. trying to unlock a mutex locked by others

Mutex types:

- PTHREAD_MUTEX_NORMAL (1) → deadlock; (2),(3) → undefined
- PTHREAD_MUTEX_ERRORCHECK error checking on all operations (hence: slower mutex): (1),(2),(3) → error
  - deadlocks involving multiple threads are still possible though
- PTHREAD_MUTEX_RECURSIVE reference counting: (1) → increase counter; (2) → decrease counter, “unlock” when the counter reaches 0, error when trying to get it below 0; (3) → error

See: `pthread_mutexattr_settype` man page.

Mutex lock/unlock

Locking/unlocking mutex is straightforward using:

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Returns: *0 on success, positive errno on error*

`*_lock` and `*_unlock` have the obvious semantics. `*_trylock` is the non-blocking equivalent of `*_lock`.

For a more fancy (and seldomly needed) feature, see also `pthread_mutex_timedlock` (timed equivalent of `*_lock`).

Mutex — example

```c
#include <pthread.h>
#include "helpers.h"

static long glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void *tmain(void *arg) {
	long n = (long) arg;
	long i, loc;
	int s;
	for (i = 0; i < n; i++) {
		s = pthread_mutex_lock(&mtx);    // begin CS
		if (s != 0)  err_sys_p(s, "pthread_mutex_lock");
		loc = glob;
		loc++;
		glob = loc;
		s = pthread_mutex_unlock(&mtx); // end CS
		if (s != 0)  err_sys_p(s, "pthread_mutex_unlock");
	}
	return NULL;
}
```

Mutex — example (cont.)

```
int main(int argc, char **argv) {        // main unchanged
        pthread_t t1, t2;
        int s;
        long n;
        if (argc != 2)
                err_quit("Usage: pthread_race ITERATIONS");
        n = atol(argv[1]);        // TODO: error checking

        s = pthread_create(&t1, NULL, tmain, (void *) n);
        if (s != 0)  err_sys_p(s, "pthread_create (t1)");
        s = pthread_create(&t2, NULL, tmain, (void *) n);
        if (s != 0)  err_sys_p(s, "pthread_create (t2)");

        s = pthread_join(t1, NULL);
        if (s != 0)  err_sys_p(s, "pthread_join (t1)");
        s = pthread_join(t2, NULL);
        if (s != 0)  err_sys_p(s, "pthread_join (t2)");

        printf("glob = %ld (expected: %ld)\n", glob, n * 2);
        exit(EXIT_SUCCESS);
} /* end of pthread_mutex.c */
```

Mutex — example (cont.)

Demo

Mutex performances

"lies, damned lies, and benchmarks", but FWIW:[2]

- 20'000'000 (20M) POSIX lock/unlock = 20.00s
- 20M SysV semop = 12.00s
- 20M flock lock/unlock = 3.46s
- pthread_mutex 10M (i.e. 20M mutex lock/unlock) = 1.65s
- pthread_race 10M = 0.15s
  - to give the wrong answer!

Note: in the specific case of pthread_mutex, mutex operations dominate over the actual critical section code, because the critical section code does almost nothing.
That's *not* the usual scenario.

[2] on Linux x86-64, Intel i5, quad-core, 2.5 GHz

Outline

1 Threads and Pthreads

2 Mutex

3 Condition variables

4 Thread safety

5 Going further

Producer-consumer problem

The producer-consumer (P-C) problem is a paradigmatic case in the theory of multi-process synchronization.

- a producer process works hard to produce discrete goods (bread, some computation result, etc). Each time a new unit is ready, the producer adds it to some shared storage containing available goods
- a consumer process works hard to consume goods. Each time he is ready, the consumer removes some available good from the shared storage

The producer-consumer problem is used to show many synchronization issues, including race conditions.

Many variants: limited/unlimited storage size, one/many producers, one/many consumers, etc.

Producer-consumer with mutex

Example (simple producer-consumer)

Unlimited storage, 1 producer, 1 consumer.

We already know how to avoid the most obvious race condition in this scenario, i.e. non-atomic, concurrent updates to the shared storage of goods.

Let’s see a naive implementation that uses Pthread mutex.

P-C with mutex — example (producer)

```c
#include <pthread.h>
#include <unistd.h>
#include "helpers.h"

static long goods = 0;  // n. of available items
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void *producer(void *arg) {      // producer thread
        int s;
        for (;;) {
                sleep(1);        // takes time to produce
                s = pthread_mutex_lock(&mtx);
                if (s != 0)  err_sys_p(s, "pthread_mutex_lock");
                goods++;
                s = pthread_mutex_unlock(&mtx);
                if (s != 0)  err_sys_p(s, "pthread_mutex_unlock");
        }
        return NULL;
}
```

P-C with mutex — example (consumer)

```
int main(int argc, char **argv) {        // consumer thread
        int s;
        pthread_t t;

        s = pthread_create(&t, NULL, producer, NULL);
        if (s != 0)  err_sys_p(s, "pthread_create");

        for (;;) {
                s = pthread_mutex_lock(&mtx);
                if (s != 0)  err_sys_p(s, "pthread_mutex_lock");
                while (goods > 0) {
                        goods--;
                        printf("consuming... good!\n");
                }
                s = pthread_mutex_unlock(&mtx);
                if (s != 0)  err_sys_p(s, "pthread_mutex_unlock");
        }

        exit(EXIT_SUCCESS);
} /* end of pc-naive.c */
```

P-C with mutex — example

Demo

Demo

- **what's wrong with this implementation?**

Demo

- **what's wrong with this implementation?**
- time-wise, the process is producer-bound: it takes time to produce, but no time (or much less) to consume
- but the consumer is polling (draining CPU, battery, etc), just to discover whether goods are available or not

Condition variables

We can solve this and similar problems with condition variables.

Intuition

A condition variable is a rendez-vous point where observers interested in the state of a shared resource can wait for some condition to be verified; complementary, others can notify waiting threads that the condition in question is verified.

The following operations can be performed *on* condition variables:

- **wait** block until a notification arrives
- **signal** send a notification

- condition variables are always used in conjunction with mutex to guarantee mutual exclusion on the shared resource
- condition variables are stateless
  - i.e. if no thread is waiting when a notification happens, the notification is lost

Creating condition variables

As with mutex, you can create condition variables either in static memory, using

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

or in dynamic memory with

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *restrict cond,
            const pthread_condattr_t *restrict attr);
```

Returns: *0 on success, positive errno on error*

See `pthread_condattr_*` syscalls for more info about condition variable attributes.

Acting on condition variables

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Returns: *0 on success, positive errno on error*

- `*_wait` does the following steps, in order:[3]
  1. unlock *mutex*
  2. suspend execution of the calling thread until *cond* is signaled
  3. lock *mutex* again

[3] we will see in a bit *why*

Acting on condition variables (cont.)

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Returns: *0 on success, positive errno on error*

- `*_signal` inform at least one of the waiting threads to wake up
- `*_broadcast` inform all waiting threads to wake up

Acting on condition variables (cont.)

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Returns: *0 on success, positive errno on error*

- `*_signal` inform at least one of the waiting threads to wake up

Use case: it doesn't matter *which* thread will perform a task, as long as one does; e.g. worker pool.

- `*_broadcast` inform all waiting threads to wake up

Use case: *all* waiting threads need to do something in reaction to the event, e.g.: publish/subscribe.

P-C with condition variables — example

Let's redo the producer-consumer example using condition variables to avoid polling.

Idea:

- a single condition variable `avail`, meaning "*goods are available*"
- if no goods are available, the consumer will wait on `avail`
- each time the producer produces something, it signals `avail` to awake the consumer (if he is asleep)

P-C with condition variables — example (producer)

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

static long goods = 0;  // n. of available items
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t avail = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {      // producer thread
        int s;
        for (;;) {
                sleep(1);        // takes time to produce
                s = pthread_mutex_lock(&mtx);
                if (s != 0)  err_sys_p(s, "pthread_mutex_lock");
                goods++;
                s = pthread_mutex_unlock(&mtx);
                if (s != 0)  err_sys_p(s, "pthread_mutex_unlock");
                s = pthread_cond_signal(&avail);
                if (s != 0)  err_sys_p(s, "pthread_cond_signal");
        }
        return NULL;
}
```

P-C with condition variables — example (consumer)

```
int main(int argc, char **argv) {        // consumer thread
        int s;
        pthread_t t;
        s = pthread_create(&t, NULL, producer, NULL);
        if (s != 0)  err_sys_p(s, "pthread_create");
        for (;;) {
                s = pthread_mutex_lock(&mtx);
                if (s != 0)  err_sys_p(s, "pthread_mutex_lock");
                while (goods == 0) {
                        s = pthread_cond_wait(&avail, &mtx);
                        if (s != 0)  err_sys_p(s, "pthread_cond_wait
                }
                while (goods > 0) {
                        goods--;
                        printf("consuming... good!\n");
                }
                s = pthread_mutex_unlock(&mtx);
                if (s != 0)  err_sys_p(s, "pthread_mutex_unlock");
        }
        exit(EXIT_SUCCESS);
} /* end of pc-cond.c */
```

P-C with condition variables — example

Demo

- no polling, by design
- consumer awaken only when needed
- CPU back under control

Wait subtleties (1)

Here is how we would like to use condition variables:

```
pthread_mutex_lock(&mtx);

if (/* shared resource is *not* in desired state */)
    pthread_cond_wait(&cond, &mtx);

/* desired state of shared resource reached;
   work on shared resource */

pthread_mutex_unlock(&mtx);
```

- shared resource access must be mutex-protected, in particular:
  - the state test
  - the actual work
- but we cannot block while holding the mutex
  - otherwise other threads won't be able to change shared resource to the state we need to wakeup → deadlock

⇒ the `unlock; wait; lock` semantics

- with atomic execution of the `wait; lock` part (why?)

Wait subtleties (2)

Let's look again at this code:

```
pthread_mutex_lock(&mtx);

if (/* shared resource is *not* in desired state */)
    pthread_cond_wait(&cond, &mtx);

/* desired state of shared resource reached;
   work on shared resource */

pthread_mutex_unlock(&mtx);
```

What if:

- other threads executing the same code are woken up together with me, but execute first?
- spurious wakeups happens? (i.e. waking up even if nobody has signaled)

You'll end up executing *as if* the condition were satisfied, while that's not necessarily the case. FAIL.

Wait subtleties (2) (cont.)

Bottom line: you should always recheck the predicate associated to a condition variable immediately after wakeup

- if it's not satisfied, wait again

Condition variable wait recipe

```
pthread_mutex_lock(&mtx);

while (/* shared resource is *not* in desired state */)
    pthread_cond_wait(&cond, &mtx);

/* desired state of shared resource reached;
   work on shared resource */

pthread_mutex_unlock(&mtx);
```

Intuition / design principle

Think of condition variables as expressing the possibility that a predicate is verified, rather than the certainty of it.

Destroying condition variable

As it happened with mutex, dynamically allocated condition variable should be destroyed when no longer needed using:

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
```

Returns: *0 on success, positive errno on error*

For your health: destroy a condition variable only when no threads are waiting on it.

Outline

1 Threads and Pthreads

2 Mutex

3 Condition variables

4 Thread safety

5 Going further

Thread safety

Definition (thread-safe function)

A function is said to be thread-safe if it can be "safely" invoked by multiple threads concurrently, i.e. without inducing race conditions, data corruption, or other synchronization problems.

Conversely, your threads should not invoke thread-*un*safe functions, without explicit synchronization among them—e.g. by using mutex to ensure that only one thread at a time invokes the function.

Thread *un*safety — example

Here is an example of a non thread-safe function, as a degenerate case of the synchronization example we've seen:

```
static int glob = 0;

void do_something(void) {
    int loc;
    /* do something useful... */
    loc = glob;
    loc++;
    glob = loc;
    /* do something else... */
}
```

- concurrent use of this function by multiple threads makes the final value of `glob` unpredictable
- common cause of *un*safety: global variables / static memory
  - precisely because they are *global*, shared resources among all threads, i.e. the basic ingredient of race conditions

Thread safety approaches

Ideally, we want all functions we use to be thread safe.
So, how do we make a function thread-safe?

1. use (and write) only reentrant functions

Definition (reentrant function)

A function is reentrant if it does not access global variables and static memory, i.e. it only accesses parameters and local variables

- local variables are allocated on the stack
- we have one-stack per thread
- no more shared resources! → no more race conditions!!

Unfortunately, that is not always possible:

- *intrinsically* global data structures
  - e.g. `malloc` & friends' lists of free/available heap memory
- old, standard functions with non-reentrant interfaces
  - e.g. returning a pointer to statically allocated memory

Thread safety approaches (cont.)

2. serialize usage of non thread-safe functions using mutex

```
static int glob = 0;
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void do_something(void) {
    int loc;
    pthread_mutex_lock(&mtx);
    /* do something useful... */
    loc = glob;
    loc++;
    glob = loc;
    /* do something else... */
    pthread_mutex_unlock(&mtx);
}
```

- similar to Java's synchronized methods
- drawback: might greatly reduce the parallelism of your program

Thread safety approaches (cont.)

3. protect critical-section snippets with mutex

```
static int glob = 0;
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void do_something(void) {
    int loc;
    /* do something useful... */
    pthread_mutex_lock(mtx);
    loc = glob;
    loc++;
    glob = loc;
    pthread_mutex_unlock(mtx);
    /* do something else... */
}
```

- drawback: hard to get right

How about functions written by others. . . ?

Thread-safe POSIX functions

POSIX requires all functions to be thread-safe, with the exception of:

asctime basename catgets crypt ctermid ctime dbm_clearerr dbm_close dbm_delete dbm_error dbm_fetch dbm_firstkey dbm_nextkey dbm_open dbm_store dirname dlerror drand48 ecvt encrypt endgrent endpwent endutxent fcvt ftw gcvt getc_unlocked getchar_unlocked getdate getenv getgrent getgrgid getgrnam gethostbyaddr gethostbyname gethostent getlogin getnetbyaddr getnetbyname getnetent getopt getprotobyname getprotobynumber getprotoent getpwent getpwnam getpwuid getservbyname getservbyport getservent getutxent getutxid getutxline gmtime hcreate hdestroy hsearch inet_ntoa l64a lgamma lgammaf lgammal localeconv localtime lrand48 mrand48 nftw nl_langinfo ptsname putc_unlocked putchar_unlocked putenv pututxline rand readdir setenv setgrent setkey setpwent setutxent strerror strsignal strtok system tmpnam ttyname unsetenv wcrtomb wcsrtombs wcstombs wctomb

- note: they *could* be thread-safe, in specific implementations
- thread-safe by default is quite nice!
- for some of them, SUSv3 introduced thread-safe equivalent
  - e.g. compare `rand`/`rand_r`, `getlogin`/`getlogin_r`, ...
- see `pthreads(7)` for more details

Outline

1. Threads and Pthreads
2. Mutex
3. Condition variables
4. Thread safety
5. Going further

Pthreads — going further

To reiterate:

> *[Pthreads is a] very large and complex API! we will only scratch the surface...*

Let's see an overview of what you're missing.

Thread-specific data & thread-local storage

Let's assume you have a function with a non thread-safe interface and you want to make it thread-safe.

Using thread-specific data you can do so without using mutex.

- thread-specific data is a standard API that allows threads to transparently use thread-specific copies of static variables

An easier to use alternative is thread-local storage. Just declare:

```
static __thread buf[BUFSIX];
```

and each thread will get its own copy of the variable.
However, this is not portable, but supported at least on Linux, Solaris, FreeBSD.

Thread cancellation

You want to do some thread orchestration where a master thread create slave threads and might need to destroy them in response to external events (e.g. user request).

Implementing this with what we've seen is cumbersome, e.g.:

- shared data structure "list of threads that should die"
- master threads add entries (e.g. thread IDs) to it
- slave threads periodically check the list and terminate voluntarily if their thread ID show up there

With thread cancellation a thread can ask another to terminate:

- either at the next synchronization point, or
- abruptly

Not particularly sane way of managing threads. . . , hard to get right.

more info

`pthread_cancel(3)`, `pthreads(7)`

Threads and signals

Don't do that.

General idea:

- signals can target individual threads or the process as a whole
- signal dispositions are global, but signal masks are per-thread
- good model: delegate signal management to a single thread, protect all other threads with masks
- bonus point: as the signal management thread has nothing else to do, make it handle signal synchronously with:

```
#include <signal.h>

int sigwait(const sigset_t *set, int *sig);
```

Returns: *0 on success, positive errno on error*

Threads vs processes

Threads, pros:

- sharing data between threads (of the same process) is easy
- thread creation is faster than process creation
  - context switch between threads *might* be faster, too

Threads, cons:

- address space is shared by default among threads → larger surface for race conditions; therefore you need:
  - judicious synchronization between threads
  - judicious use of thread-safe functions
- poor isolation: a bug in one thread can very easily damage the state of other threads
- threads compete for virtual address space
  - thread stacks and thread-local storage can deplete available virtual addresses (especially on 32bit architectures)
  - this is not the case for processes, as each process has its own address space