Threads (vs Processes)

Computer runs multiple processes concurrently (interleaved time-slicing and/or multicore parallel).

Multiple threads too, with different overhead and use case:

	process	thread
overhead	higher	lower
data sharing with relatives	unshared	shared

Threads are great for concurrency over shared data.

Concurrency, Parallelism

Related but not synonym.

Parallelism: You really insist k cores working on k subproblems at the same time to finish in $1/k$ of the time. You don't accept interleaving on 1 core.

Concurrency: Neutral on parallelism vs interleaving. Just means structuring your code as k workflows, they're independent apart from a bit of communication or shared data.

Parallelism requires concurrent coding. Converse not true:

Concurrent coding could also mean you don't mind interleaving, you just find concurrency a better code structure for your task, you would rather the OS interleave for you than you interleave by hand.

Thread Creation

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

Returns 0 if success, +ve error code if error.

`tid`: address to store thread ID.

`start`: [pointer to] function like this:

```
void* mythreadstart(void *myarg);
```

New thread starts from it, `myarg` gets `arg`.

`attr`: `NULL` for default: priority, scheduling, other technical options.

threads.c

threads-retval.c (return value example)

`pthread_t` Portability Notes

`pthread_t` could be number or pointer or struct depending on system. No portable way to print, not even ==. Instead use:

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

On Linux, `unsigned long` holding address. May be acceptable to print as such when debugging.

For additional confusion/fun: Linux has `gettid` giving yet another number (`pid_t`, closely related to PIDs).

Wait (Join) for Thread Termination

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

Wait for thread termination, optionally get return value (if `retval` not `NULL`).

Any thread can wait for any thread, no parent-child relation required. (In fact no parent-child relation at all.)

Zombie thread happens if a thread terminates but you don't join it.

Thread Termination

A thread terminates if one of:

Thread's start function returns.

Thread calls:
`void pthread_exit(void *retval);`

Something cancels the thread:
`int pthread_cancel(pthread_t thread);`
(Returns immediately. Use `pthread_join` to wait. Target thread can postpone termination until sensible point.)

Main thread `main` function returns, or something calls `exit`. All threads die.

Pitfall of Data Sharing: Race Conditions

As simple as incrementing a counter:

1. read
2. compute new value
3. write

blows up if 2 threads A and B do it interleavedly:

1. A's turn, read, get 5
2. B's turn, read, it's still 5
3. A's turn, compute 6
4. still A's turn, write 6
5. B's turn, compute 6
6. still B's turn, write 6

race.c

Synchronization Primitive 1: Mutex

Mutex = “mutual exclusion”.

A thread can request to “acquire/lock” a mutex. Once acquired, can (should) later “release/unlock” it.

Until released, other threads requesting to acquire are blocked. When released, one requester is chosen to acquire it. Etc.

Workflow for accessing shared data:

1. (Request to) Acquire mutex:
   ```
   int pthread_mutex_lock(pthread_mutex_t *m);
   ```
2. Work on shared data. (“critical section”)
3. Release mutex:
   ```
   int pthread_mutex_unlock(pthread_mutex_t *m);
   ```

Now threads can access shared data in an orderly fashion.

Pthreads Static And Dynamic Mutex

Static mutex: Pthreads offers simplified creation if your mutex is global:

```
pthread_mutex_t my_mutex =
    PTHREAD_MUTEX_INITIALIZER;
```

and no need to cleanup. mutex-static.c

Dynamic mutex: If your mutex space is local var or malloc'ed, better init and cleanup by:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *mutex_attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

mutex-dynamic.c

Synchronization Primitive 2: Condition Variable

Sometimes a thread wants to idle until another thread has changed shared data.

Example: A sender thread puts a message in a shared message box. A receiver thread is waiting for message to appear.

Condition variable = medium for sender to say “updated, please check now”.

```
int pthread_cond_signal(pthread_cond_t *cond);
```

and for receiver to wait for that:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

Reason for the mutex is explained next.

Why `pthread_cond_wait` needs mutex

Basic idea for receiver workflow:

1. Acquire mutex (because next is check shared data).
2. Check message box. Suppose no message, want to wait:
3. Release mutex (sender will need it).
4. Start waiting on cond var.
5. The wait ends.
6. Re-acquire mutex (because next is re-check).
7. Goto 2.

But race condition if 3–4 or 5–6 is interleavable with sender thread or other receiver threads.

Want them atomic (inseparable). That's `pthread_cond_wait`, needing both cond var and mutex.

Pthreads Static And Dynamic Cond Var

Static cond var: Pthreads offers simplified creation if your cond var is global:

```
pthread_cond_t my_cond = PTHREAD_COND_INITIALIZER;
```

and no need to cleanup. cond-static.c

Dynamic cond var: If your cond var space is local var or malloc'ed, better init and cleanup by:

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

cond-dynamic.c

General Workflow for Complex Use Case

In a complex application: Different waiters wait for different predicates, updater unsure an update is relevant to whom.

Simple workflow that always works (just a bit inefficient):

Updater notifies everyone:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Waiter checks and anticipates to wait again:

```
pthread_mutex_lock(&m);
while (! my_predicate) {
  pthread_cond_wait(&c, &m);
}
// now my turn to work, and then:
pthread_mutex_unlock(&m);
```

cond-static-broadcast.c