Thread programming with POSIX Threads (Pthreads)

Karl Ljungkvist

What are threads?

Definition:

- Independent streams of instructions within a single program, which can be scheduled independently by the OS

In practice:

- A thread is a procedure/function running independently from the main program.
- A way of utilizing multiple cores.

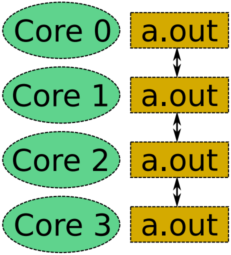
Lightweight process:

- Only duplicates a necessary minimal
- Most resources are shared within the process
- Less overhead

Processes vs Threads

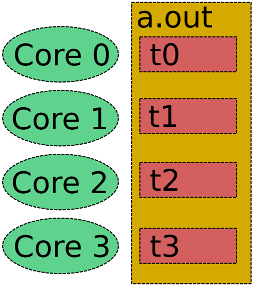
Processes:

- Multiple instances of the same program, which can communicate by message passing



Threads:

- Single program with parallel internal threads sharing resources (memory, open files, etc)



Shared resources

A thread contains private

- Program counter
- Registers and stack pointer
- Scheduling properties (i.e. policy and priority)
- Set of pending and blocked signals

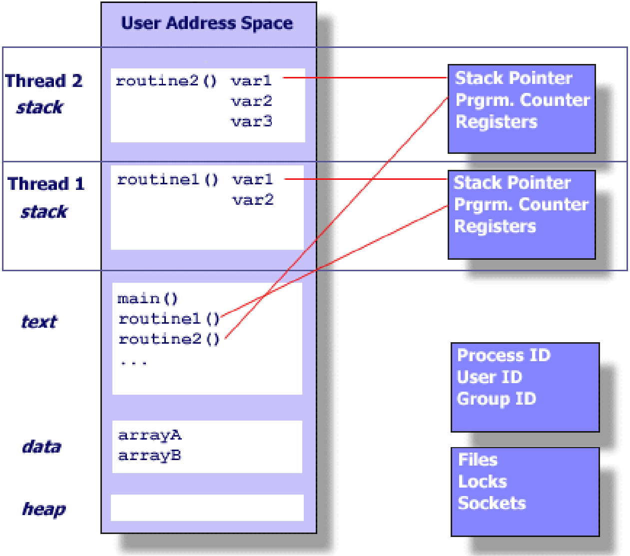
In addition, a Unix process also has

- Process, process group, user, and group IDs
- Environment
- Working directory
- Program instructions
- Stack and Heap
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools

Lightweight process

- Process to thread creation overhead $\sim$ 10:1

Two threads in a process



Question

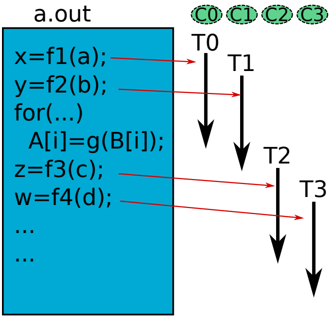
What is private to each thread in a process?

- Program counter
- Address space
- Stack pointer
- Registers
- Open files

Usage of threads

Traditionally (single-core processor):

- Overlapping CPU work with I/O: Reading from file mostly involves waiting for disk. Let another thread to do useful work in the meantime.
- Priority/real-time scheduling: Prioritize important tasks (also, allows multi-user system).
- Asynchronous event handling: Unpredictable events, e.g. web server requests, can be serviced by starting dedicated thread

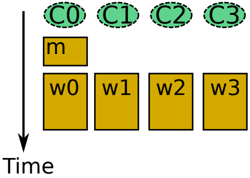


Today:

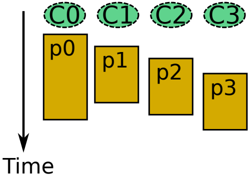
- Used to perform tasks in parallel on a multi-core system.

Thread programming models

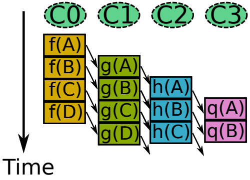
Manager-worker
Single manager thread assigns work to a set of worker threads. Typically used for a dynamic pool of tasks with irregular work load.



Peer
Similar to the manager/worker model, but the main thread participates in the work. Typically used for static homogeneous tasks.



Pipeline
Like a car assembly line; a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread.



Why threads?

Benefits over processes:

- Less overhead from creation
- Shared resources
  - Threads can simply read each others memory
  - Changes by one thread (e.g. closing a file) is visible to others
  - No communication is needed
  - Same address space $\Rightarrow$ pointers with same value point to same address
  - However, 'interesting' implications (see below)

Question

Why do we use threads?

- To utilize multi-cores
- Less overhead than processes
- More error-proof than processes
- Simpler communication than processes

Consequences of shared resources

There is a price:

- All threads can access shared resources, e.g. memory
- Like a shared notepad; other users (threads) might not be done writing
- Need a way to synchronize access
- Also, need explicit synchronization to maintain algorithm integrity. For example, cannot start next pipeline stage before the previous one is done.

Compare

- This was not a problem for MPI/processes
- Explicit message passing introduces synchronization

Example

Context:

- Tim and Tom share a fridge
- There should always be milk
- Policy: If no milk in fridge, buy milk

Example case:

Tim	Tom
Comes home, no milk in fridge	
Goes to the store	
Buys milk	Comes home, no milk in fridge
(still buys milk, store is crowded)	Goes to the store
Returns, puts milk in fridge	Buys milk
	Returns, fridge already contains milk

Race condition:

- Unexpected result

⇒ Solution: Put a lock on the fridge...

Race conditions

Problem:

- Threads scheduled by operating system
- Instructions in threads might be interleaved arbitrarily
- Partial results read/overwritten
- No guarantee on ordering of operations

Solution:

- Need to prevent instruction-level interleaving in *critical section*, i.e. code of sensitive operation
- *Atomic* – indivisible, performed entirely or not at all

Locks:

- A way of achieving atomicity:
- Only one thread at a time can claim the lock

⇒ Only one thread at a time can be in the critical section

```
int a_shared; //shared
lock_t l; //shared
...
lock(l);
  int a = a_shared;
  a = fun(a);
  a_shared = a;
unlock(l);
```

Question

Account deposit

- Shared variable `account`, start value 1000
- Two threads depositing to `account`:

Thread 1:	Thread 2:
`account += 200`	`account += 500`

- Assumption: '+=' operation not atomic

What are the possible results?

- 1200
- 1500
- 1700
- Something else

Pthreads

POSIX Threads (Pthreads)

P**ortable **O**perating **S**ystem **I**nterface for UNI**X

- Portable standard for thread programming, specified by the IEEE POSIX 1003.1c standard (1995)
- C Language
- Supported by most operating systems: Linux, Mac OS X, Windows (partially), and others

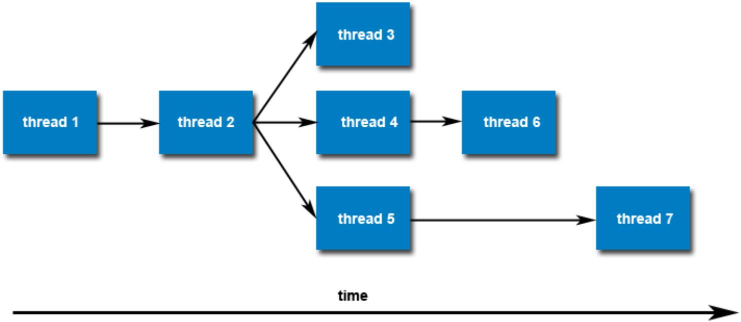
The Pthreads API contains over 60 subroutines which can be grouped into three major classes:

- ***Thread management***: creating, terminating, joining
- ***Mutexes***: provides exclusive access to code segments and variables with the use of locks (mutual exclusion)
- ***Condition variables***: provides synchronization and communication between threads that share a mutex

Creating and terminating threads

`pthread_create (ptr, attr, func, arg)`

- Creates a thread which starts running the specified function.
  Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.



Creating and terminating threads

There are several ways in which a Pthread may be terminated:

- The thread returns from its starting routine
- The thread makes a call to `pthread_exit()`
- The thread is canceled by another thread via the `pthread_cancel()` routine
- The entire process is terminated, i.e., `main()` finishes without self calling `pthread_exit()`

Note: By calling `pthread_exit()` also in `main()`, i.e., on the master thread, all threads are kept alive even though all of the code in `main()` has been executed. Can also do explicit wait with `pthread_join()`

Example: Hello World

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *HelloWorld(void *arg){
    printf("Hello World!\n");
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int t;
   for(t=0; t<NUM_THREADS; t++)
      pthread_create(&threads[t],
                     NULL,
                     HelloWorld,
                     NULL);
   pthread_exit(NULL);
}
```

Question: How many threads will run when executing this program?

- 1
- 4
- 5
- 6

Passing arguments

Note, can only pass one argument of type `void*`. Use structs and type cast to `void*`

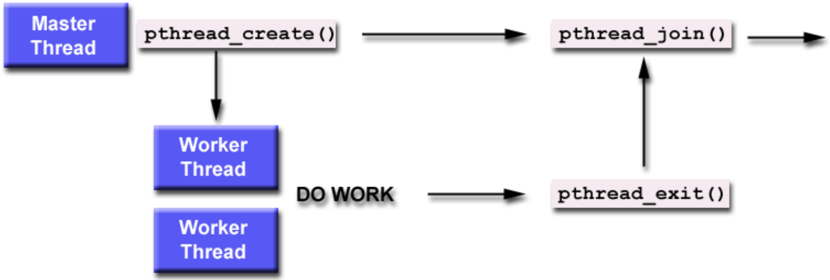
```
struct thread_data{
  int field1;
  double field2};

void *HelloWorld(void *arg){
  struct thread_data *my_data = (struct thread_data*) arg;
  int f1 = my_data->field1;
  double f2 = mydata->field2;
  ... }

int main (int argc, char *argv[]){
  ...
  struct thread_data data
  data.field1=5; data.field2=3.14;
  pthread_create(&threads[t], NULL, HelloWorld, (void*)&data);
  ...
```

See example `hello_arg2.c` in lab.

Joining threads (waiting)

`pthread_join (thread, status)`

Blocks the calling thread until the specified thread terminates.



When a thread is created, its attribute must be set to joinable (default for most Pthreads implementations).

Joining threads (waiting)

To explicitly create a thread as joinable:

- Declare a pthread attribute variable of the `pthread_attr_t` data type
- Initialize the attribute variable with `pthread_attr_init()`
- Set the attribute detached status with `pthread_attr_setdetachstate()`
- *Create, use, and terminate thread*
- When done, free library resources used by the attribute with `pthread_attr_destroy()`

Example: join (join.c)

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for (t=0; t<NUM_THREADS; t++)
  pthread_create(&thread[t],&attr,func,(void *)&data);

pthread_attr_destroy(&attr);

for (t=0; t<NUM_THREADS; t++)
  pthread_join(thread[t], &status);
```

Can also set the state to `PTHREAD_CREATE_DETACHED` (Default value is joinable.)

Other attributes that can be set are stacksize and scheduling policy. (For more info see Pthreads manual.)

Global and local data

Data allocated on the stack, i.e., within functions, is local and private to the threads. All other data is global.

```
// Global data accessible to all threads
int GlobData[Nsize];

void *threadfunc(void *arg){
  // Local data private to the calling thread
  int LocData[Nsize];
  int *array = (int *) arg;
  ...
}

int main(int argc, char *argv){
  // Global data but needs to be passed to threads
  int GlobData2[Nsize];
  ...
  pthread_create(&thread, NULL, threadfunc, (void *) GlobData2);
  ...
}
```

See data.c

Question

What does `pthread_join` do?

- Waits for another thread to finish
- Merges two given threads
- Synchronizes all running threads

Mutexes

Recall *Race condition*:
Unsynchronized parallel writes to unprotected memory will give unpredictable results.

Mutex variables

- Lock variables
- *Mutual exclusion*
- One of the primary means of implementing thread synchronization and for protecting shared data.

Mutexes

A typical sequence in the use of a mutex is as follows:

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- Only one succeeds and that thread owns the mutex
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally the mutex is destroyed

When several threads compete for a mutex, the losers block at that call - an unblocking call is available with "trylock" instead of the "lock" call. (Trylock is much faster, it does not block but it also does not have to deal with queues of multiple threads waiting on the lock.)

Mutex functions

```
pthread_mutex_init (mutex, attr)
pthread_mutex_destroy (mutex)
phtread_mutex_lock (mutex)
pthread_mutex_trylock (mutex)
pthread_mutex_unlock (mutex)
```

The mutex attribute can be set to:

- `PTHREAD_MUTEX_NORMAL_NP`
- `PTHREAD_MUTEX_RECURSIVE_NP`
- `PTHREAD_MUTEX_ERRORCHECK_NP`

Or just use `attr=NULL` for default values.

Example: mutex (mutex.c)

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5
pthread_mutex_t mutexsum;
int sum=0;

void *addone(void *arg){
  pthread_mutex_lock (&mutexsum);
  sum += 1;
  pthread_mutex_unlock (&mutexsum);
  pthread_exit(NULL);}

int main (int argc, char *argv[]){
  ...
  pthread_mutex_init(&mutexsum, NULL);
  for(t=0; t<NUM_THREADS; t++)
    pthread_create(&threads[t], NULL, addone, NULL);
  for (t=0; t<NUM_THREADS; t++)
    pthread_join(thread[t], &status);
  printf("Sum = %d\n",sum);
```

Condition variables

Example: wait for a condition
Two threads, one must wait for the other to complete some operations.

```
volatile int cond = 0; // suspect to unpredictable changes
```

thread 1:

```
... (do stuff)
cond = 1;
```

thread 2:

```
while(cond==0);
... (do stuff)
```

Problem:
Thread 2 will *busy wait* – CPU cycles wasted.

Need a way to let thread 2 sleep, and a way of letting thread 1 waking it up...

$\Rightarrow$ Condition variable

Condition variables

A condition variable is used for synchronization of threads. It allows a thread to block (sleep) until a specified condition is reached.

```
pthread_cond_init (cond, attr)        – use attr=NULL
pthread_cond_destroy (cond)
phtread_cond_wait (cond, mutex)       – block thread
pthread_cond_signal (cond)            – wake one thread
pthread_cond_broadcast (cond)         – wake all threads
```

A condition variable is always used in conjunction with a mutex lock. Proper locking and unlocking of the associated mutex variable is important.

Condition variables

pthread_cond_wait():

```
pthread_mutex_lock(mutexvar);

if (!condition)
  pthread_cond_wait(condvar,mutexvar);

pthread_mutex_unlock(mutexvar);
```

`pthread_cond_wait()` blocks a thread until the condition variable is signaled. It will automatically release the mutex while it waits. After the thread is awakened, mutex will be automatically locked for use by the thread.

Note, wait does not use any CPU cycles until it is woken up (`mutex_lock` uses CPU cycles for polling)

Condition variables

pthread_cond_signal(), pthread_cond_broadcast():

```
pthread_mutex_lock(mutexvar);

if (condition)
  pthread_cond_signal(condvar);

pthread_mutex_unlock(mutexvar);
```

`pthread_cond_signal()` is used to wake up another thread which is waiting on the condition variable. It should be called after `mutexvar` is locked, and must unlock `mutexvar` in order for `pthread_cond_wait()` to complete.

If more than one thread is in a blocking wait, use `pthread_cond_broadcast()` to wake all.

Questions

Question: Why do we need condition variables?

- To avoid race conditions
- To prevent threads from busy waiting
- To achieve atomicity

Question: Why do we need a mutex with a condition variable?

- To allow threads to awake owning a mutex
- To include the wait command in a critical section
- To control which threads can call signal

Example: barrier

```
pthread_mutex_t lock;
pthread_cond_t signal;
int waiting=0, state=0;

void barrier(){
  pthread_mutex_lock (&lock);
  int mystate=state;
  waiting++;
  if (waiting==nthreads){
    waiting=0; state=1-mystate;
    pthread_cond_broadcast(&signal);}
  while (mystate==state)
    pthread_cond_wait(&signal,&lock);
  pthread_mutex_unlock (&lock);
}
```

Note: use while-statement since spurious wake ups of threads sleeping in wait may occur.

Barrier (2)

pthread_barrier_t:

- Not part of standard
- Still, supported by most implementations

pthread_barrier_init	(barrier, attr, nthr)	– create barrier
pthread_barrier_destroy	(barrier)	– destroy barrier
pthread_barrier_wait	(barrier)	– all nthr threads wait

Example:

```
pthread_barrier_t bar;
pthread_barrier_init(&bar, NULL, nthreads)
pthread_barrier_wait(&bar);
```

Example: Enumeration sort

Sort an array of numbers:

```
for (j=0;j<len;j++)
{
  rank=0;
  for (i=0;i<len;i++)
    if (indata[i]<indata[j]) rank++;
  outdata[rank]=indata[j];
}
```

Parallelization idea:
For each element (j) count how many other elements (i) are smaller than it.
$\Rightarrow$ Perfectly parallel since all the (j) iterations are independent.

Example: Enumeration sort

Solution 1: (enumsort.c)

For each task (element) start a new thread, but start only `NUM_THREADS` threads at a time.

```
for (int j=0;j<len;j+=NUM_THREADS){ /* Manager */

  for(int t=0; t<NUM_THREADS; t++){
    el=j+t;
    pthread_create(&threads[t],&attr,findrank,(void*)el);}

  for(int t=0; t<NUM_THREADS; t++)
    pthread_join(threads[t], &status);
}
```

```
void *findrank(void *arg){ /* Worker */
  int rank=0;long j=(long)arg;

  for (int i=0;i<len;i++)
    if (indata[i]<indata[j]) rank++;

  outdata[rank]=indata[j];
  pthread_exit(NULL);}
```

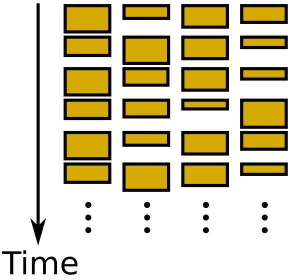
(Q) What's the problem?

- Race condition
  – incorrect result
- Too small tasks
  – bad performance

Example: Enumeration sort

Solution 1:

- Little work per task
- High overhead in creating and terminating threads
- Also, lots of synchronization points



Solution 2:
Define larger tasks, let each task be to count the rank of *len/nthreads* elements $\Rightarrow$ only one task per thread and totally nthreads tasks. Minimal synchronization and thread management overheads.

Example: Enumeration sort

Solution 2: (enumsort2.c)

```
void *findrank(void *arg){
  int j1 = ((struct index_t *) arg).j1;
  int j2 = ((struct index_t *) arg).j2;
  for (j=j1;j<j2;j++){
    int rank=0;
    for (i=0;i<len;i++)
      if (indata[i]<indata[j]) rank++;
    outdata[rank]=indata[j];}
}
```

```
struct index_t {
  int j1, j2;
};
```

```
int chunksize=len/NUM_THREADS;
for (t=0; t<NUM_THREADS-1; t++)
{
  index[t].j1=t*chunksize; index[t].j2=(t+1)*chunksize;
  pthread_create(&threads[t], &attr, findrank, (void *)&index[t]);
}
index[t].j1=t*chunksize; index[t].j1=(t+1)*chunksize;
findrank((void *)&index[NUM_THREADS-1]);

for(t=0; t<NUM_THREADS-1; t++)
  pthread_join(threads[t], &status);
```

Example: Reader/writer problem

Situation:

- Shared data, which several threads access
- Some threads are writers, update memory
- Other are readers, don't change data

Protection needed:

- Read-write lock:
  - `read_lock`
  - `read_unlock`
  - `write_lock`
  - `write_unlock`

Example: Reader/writer problem

First attempt:

```
volatile int num_readers = 0;
volatile int writer_here = 0; /* 0 or 1 */

void read_lock() {
  while (writer_here);
  num_readers++;
}

void read_unlock() {
  num_readers--;
}

void write_lock() {
  while (writer_here || num_readers > 0);
  writer_here = 1;
}

void write_unlock() {
  writer_here = 0;
}
```

- All fine?

No,

- Simultaneous `read_lock` and `write_lock` can succeed!
- Only allow one thread to access the state simultaneously

Example: Reader/writer problem

Second attempt:

```
volatile int num_readers = 0;
volatile int writer_here = 0;
pthread_mutex_t mtx;

void read_lock() {
  pthread_mutex_lock(mtx);
  while (writer_here);
  num_readers++;
  pthread_mutex_unlock(mtx);
}
void read_unlock() {
  pthread_mutex_lock(mtx);
  num_readers--;
  pthread_mutex_unlock(mtx);
}
```

```
void write_lock() {
  pthread_mutex_lock(mtx);
  while (writer_here ||
         num_readers > 0);
  writer_here = 1;
  pthread_mutex_unlock(mtx);
}
void write_unlock() {
  pthread_mutex_lock(mtx);
  writer_here = 0;
  pthread_mutex_unlock(mtx); }
```

Better?

- Deadlock: can get stuck in while loop
- Waiting thread can't hold the mutex

Example: Reader/writer problem

Third attempt:

```
volatile int num_readers = 0;
volatile int writer_here = 0;
pthread_mutex_t mtx;

void read_lock() {
  int success=0;
  while(!success) {
    pthread_mutex_lock(mtx);
    if(writer_here)
      pthread_mutex_unlock(mtx);
    else success=1;
  }
  num_readers++;
  pthread_mutex_unlock(mtx);
}
void read_unlock() {
  pthread_mutex_lock(mtx);
  num_readers--;
  pthread_mutex_unlock(mtx); }
```

```
void write_lock() {
  int success=0;
  while(!success) {
    pthread_mutex_lock(mtx);
    if (writer_here ||
        num_readers > 0)
      pthread_mutex_unlock(mtx);
    else success=1;
  }
  writer_here = 1;
  pthread_mutex_unlock(mtx);
}
void write_unlock() {
  pthread_mutex_lock(mtx);
  writer_here = 0;
  pthread_mutex_unlock(mtx); }
```

OK?

- Incorrect, need two mutexes
- Wasteful

Example: Reader/writer problem

Final attempt:

```
volatile int num_readers = 0;
volatile int writer_here = 0;
pthread_mutex_t mtx;
pthread_cond_t cond_r, cond_w;

void read_lock() {
  pthread_mutex_lock(mtx);
  while(writer_here)
    pthread_cond_wait(cond_r);
  num_readers++;
  pthread_mutex_unlock(mtx);
}
void read_unlock() {
  pthread_mutex_lock(mtx);
  num_readers--;
  if (num_readers==0)
    pthread_cond_signal(cond_w);
  pthread_mutex_unlock(mtx); }
```

```
void write_lock() {
  pthread_mutex_lock(mtx);
  while (writer_here ||
         num_readers > 0)
    pthread_cond_wait(cond_w);
  writer_here = 1;
  pthread_mutex_unlock(mtx);
}
void write_unlock() {
  pthread_mutex_lock(mtx);
  writer_here = 0;
  pthread_cond_signal(cond_w);
  pthread_cond_broadcast(cond_r);
  pthread_mutex_unlock(mtx); }
```