POSIX threads (pthreads)

POSIX threads: pthreads

- POSIX: Portable Operating System Interface for UNIX – Interface to Operating System utilities
- Pthreads: The POSIX threading interface
  - Thread implementations adhering to the POSIX standard
  - System calls to create and synchronize threads
  - Should be relatively uniform across UNIX-like OS platforms
- Pthreads contain support for
  - Creating parallelism
  - Synchronization
- No explicit support for communication, because shared memory is implicit

Creating and terminating pthreads

`pthread_create(thread,attribute,thread_function,arg)`

- `thread`: unique identifier for the new thread (thread id or handle)
- `attribute`: object that may be used to specify attribute
  - example attribute: minimum stack size
- `thread_function`: function to be executed when thread is created
- `arg`: argument to be passed to thread_function when it starts
- returns `error_code`: set to nonzero if the create operation fails

`pthread_exit(status)`, `pthread_attr_init(attr)`, `pthread_attr_destroy(attr)`

Example: creating and terminating pthreads

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     4

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   /* Last thing that main() should do */
   pthread_exit(NULL); /* main will block and wait until all threads are done */
}
```

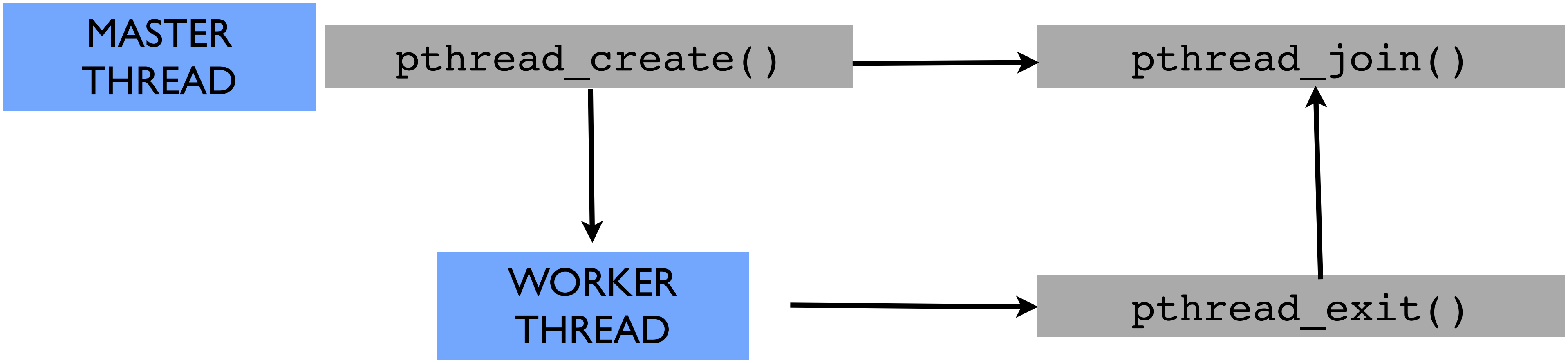
Example: thread argument passing

```
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &thread_data_array[t]);
   ...
}
```

- *each thread receives a unique instance of the data structure*

Thread management

- `pthread_join(thread_id, status)`
  - blocks the calling thread until the specified thread_id terminates
- `pthread_yield()`
  - Informs the scheduler that the thread is willing to yield its quantum, requires no argument
- `pthread_detach(thread);`
  - Informs the library that the thread's exit status will not be needed by subsequent `pthread_join` calls, resulting in better thread performance.



pthread attributes

- Once an initialized object attribute exists, changes can be made.
  - To change the stack size of a thread to 8192 (before calling pthread_create):
    - `pthread_attr_setstacksize(&my_attributes, (size_t)8192);`
  - To get the stack size:
    - `size_t my_stack_size; pthread_attr_getstacksize(&my_attributes, &my_stack_size);`

Other pthread attributes

- Detached state: set if no other thread will use pthread_join to wait for this thread (improves efficiency)
- Guard size: use to protect against stack overflow
- Inherit scheduling attributes (from creating thread) – or not
- Scheduling parameter(s) – in particular, thread priority
- Scheduling policy: FIFO or Round Robin
- Contention scope: with what threads does this thread compete for a CPU
- Stack address: explicitly dictate where the stack is located
- Lazy stack allocation: allocate on demand (lazy) or all at once, "up front"

Shared data

- Variables declared outside of main are shared
- Objects allocated on the heap may be shared (if pointer is passed)
- Variables on the stack are private: passing pointer to these to other threads can cause problems

Loop-level parallelism

- Many applications have parallelism in loops

```
double stuff [n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    ...
    pthread_create (..., update_stuff,...,&stuff[i][j]);
```

- But overhead of thread creation is nontrivial
    - update_stuff should have a significant amount of work
- Common performance pitfall: too many threads
    - Cost of creating a thread is 10s of thousands of cycles on modern architectures
    - Solution: "threadblocking": use a small # of threads, often equal to the number of cores/processors or hardware threads
        - "block" of loop iterations executed by each thread

Example of parallel execution: "blocked" threads

thread 0	thread 1	thread 2	thread 3
`i` = 0 … 127	`i` = 128 … 255	`i`= 256 … 383	`i`= 384 … 511
`A[i]`	`A[i]`	`A[i]`	`A[i]`
`+`	`+`	`+`	`+`
`B[i]`	`B[i]`	`B[i]`	`B[i]`
`=`	`=`	`=`	`=`
`C[i]`	`C[i]`	`C[i]`	`C[i]`

Data races

- Common correctness pitfall: race conditions
- A race condition or data race occurs when:
  - Two threads access the same variable, and at least one does a write
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

```
static int s = 0;
```

```
// Thread 1
for (i=0; i<n/2; i++)
    s = s + 1
```

```
// Thread 2
for (i=n/2; i<n; i++)
    s = s + 1
```

Example of data race

- Increments may be lost if both threads read simultaneously
- Need to make the read+increment+write an "atomic" operation
- Solution: locks aka Mutexes

```
static int s = 0;
```

```
// Thread 1 (core 1)
read s into R1
increment R1
write R1 to s
```

```
// Thread 2 (core 2)
read s into R2
increment R2
write R2 to s
```

Basic synchronization: mutexes

- Mutexes -- mutual exclusion aka locks
  - threads are working mostly independently
  - need to access common data structure

```
lock_t *lock = alloc_and_init();
acquire(lock);
access data
release(lock);
```

Mutexes in pthreads

- To create a mutex:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&amutex, NULL);
```

- To use it:

```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- To deallocate it:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Correctness pitfalls with locks

- Locks cover too small a region

```
acquire (a)
tmp = s
release (a)
tmp++                    /* updates can still be lost */
acquire (a)
s = tmp
release (a)
```

- Multiple locks may be held, but can lead to deadlock:

```
thread1            thread2
lock(a);           lock(b);
lock(b);           lock(a);
```

Performance pitfalls with locks

- Critical region is too large
  - Little or no true parallelism
  - Lock cost can go up with more contention
  - Solution: make critical regions as small as possible (but no smaller)
  - Solution: Use different locks for different data structures
- Locks themselves may be expensive
  - The overhead of locking / unlocking can be high

Basic synchronization mechanisms: barriers

- Barrier -- global synchronization
- Especially common when running multiple copies of the same function in parallel (SPMD)
- simple use of barriers -- all threads hit the same one

```
work_on_my_subgrid();
barrier;
read_neighboring_values();
barrier;
```

- more complicated -- barriers on branches (or loops)

```
if (thread_id % 2 == 0) {
  work1();
  barrier;
} else {
  barrier;
}
```

- barriers are not provided in all thread libraries

Creating and initializing a barrier

- To (dynamically) initialize a barrier, use code similar to this (sets the number of threads to 4):

```
pthread_barrier_t b;
pthread_barrier_init(&b,NULL,4);
```

- The second argument specifies an object attribute; using NULL yields the default attributes.
  - other attributes: `PTHREAD_PROCESSOR_SHARED/` `PTHREAD_PROCESSOR_PRIVATE`
- To wait at a barrier, a process executes:

```
pthread_barrier_wait(&b);
```

Summary of pthreads

- POSIX Threads are based on OS features
  - can be used from multiple languages (need appropriate header)
  - familiar language for most of program
  - ability to shared data is convenient
- Pitfalls
  - data race bugs are hard to find because they can be intermittent
  - deadlocks are usually easier, but can also be intermittent
- OpenMP is commonly used today as an alternative