Introduction to Posix threads

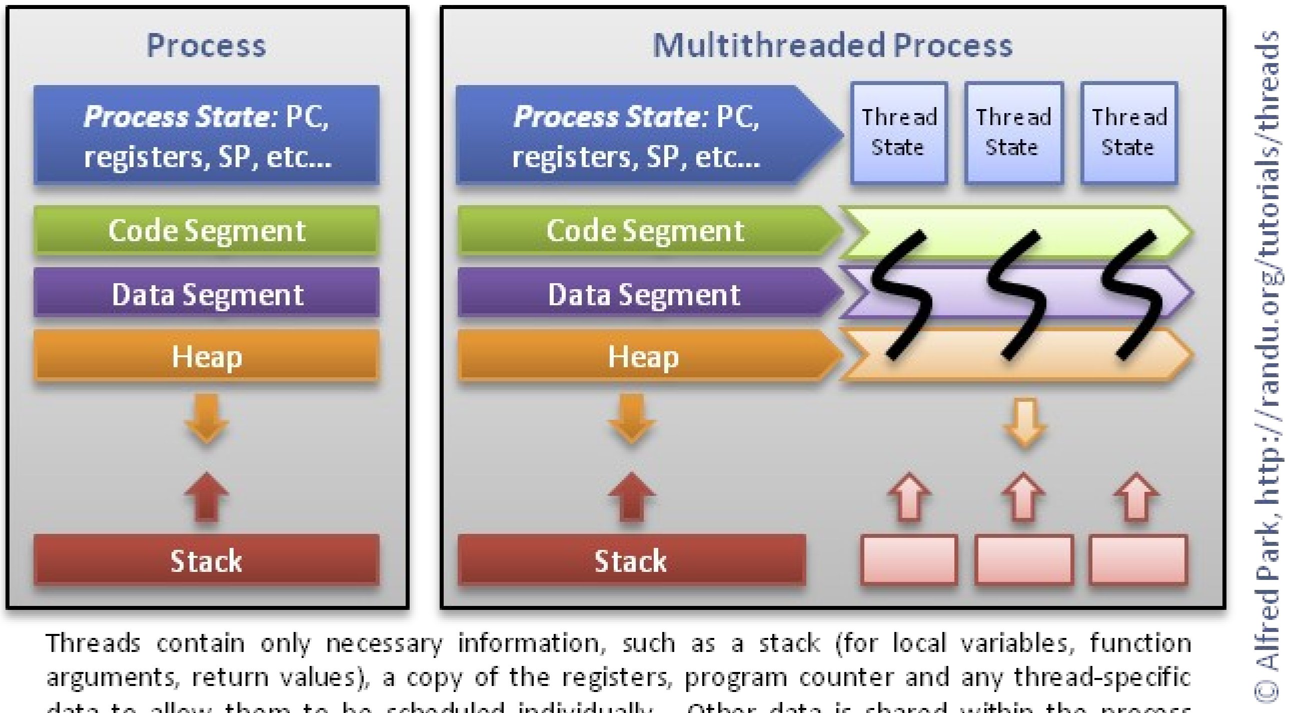
What is a thread?

- A sequence of machine instructions.
- The smallest unit of processing that a scheduler works on.

Threads and processes -

- A process (created by a fork() command) can have multiple threads of execution which are executed asynchronously.
- Every process has at least one thread

Single thread vs many threads in a process



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

Differences between a thread and a process

- Processes do not share their memory address space while threads executing within the same process share the address space.
- Processes execute independently of each other. The synchronization between processes is taken care of by the operating system. Thread synchronization is taken care of by the process which is executing the threads.
- Context switching between threads is fast as compared to context switching between processes.
- Interaction between two processes is achieved through inter-process communication. Threads executing under the same process can communicate easily as they share most of the resources.

Threads in a process share many resources

- Process instructions – all threads can access all instructions
- Global data
- open files (descriptors) that have been opened prior to the thread creation
- signals and signal handlers
- current working directory
- User and group id

Unique attributes of a thread

- Thread ID which is different from the pid. It is of type pthread_t
- A set of registers and a stack pointer
- A stack for variables local to the thread and return addresses
- A thread specific signal mask
- The thread priority

How to create threads in a process

A process always has a default thread. To create additional threads

```
#include <pthread.h>

int pthread_create(pthread_t *tidp, const pthread_attr_t *attr, void
*(*start_func)(void), void *arg)
```

1) A pointer to the thread ID.
2) What type of thread. A default type can be created by setting to NULL.
3) The name of the function to execute which is a pointer.
4) The arguments to the function that the thread will execute.

How a thread terminates

- If the thread uses return() from its starting function.
- If it is canceled by some other thread. The function used in that case is pthread_cancel().
- If it calls pthread_exit() function from within itself.

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

- The return value is accessed by another thread which should be waiting for this thread to terminate.

How one thread waits for another thread

```
#include <pthread.h>

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

- The thread that issues pthread_join() is suspended until the other thread finishes execution.
- The waiting thread can choose to wait for a specific thread whose thread ID is given.
- The waiting thread can receive the return value of the exiting thread. If that information is not wanted, the second parameter is set to NULL.

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

#define NUM_THREADS 6

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

/* thread function */
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}

int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];

  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

Similar code using Qthreads in C++

```
#include <iostream>
#include <Qthread>


using namespace std;

class MyThread: public Qthread
{
   private:
          int tid;
   public:
          MyThread (int i) : ID (i)
          {}
          void run()
          {   cout << "Thread " << ID << " is running\n": }
};

int main(int argc, char *argv[]) {
  int N = atoi (argv[1]);
  myThread *x[n];

  /* create threads */
  for (int i = 0; i < N; ++i) {
      x[i] = new MyThread(i);
      x[i]->start();
  }

  /* block until all threads complete */
  for (int i = 0; i < N; ++i) {
      x[i]->wait();
  }

  return EXIT_SUCCESS;
}
```

Some other useful pthread functions

How a thread finds its own ID:

```
pthread_t pthread_self(void);
```

The pthread_self() function returns the ID of the calling thread. This is the same value that is returned in the first parameter in the pthread_create() call that created this thread.

Tell another thread to terminate:

```
int pthread_cancel(pthread_t tidx);
```

One thread tells the thread whose ID is tidx to terminate. Important in tree-based algorithms when a solution has been found and all other threads can stop executing.

Threads and concurrency

On a multicore computer, multiple threads can be used to perform a calculation in parallel:

- Different threads can run independent tasks in parallel - functional parallelism.
- Different threads can perform the same task on different data – data parallelism.

Issues to be considered:

- Is there shared data? If so, how is it protected?
- Are the threads accessing a shared resource and how is the access coordinated?

How to create a non-joinable thread

A default thread's resources are not released until another thread calls pthread_join().

In some cases, a thread needs to be able to run independently of all other threads. This can be accomplished by “detaching” the thread:

```
int pthread_detach(pthread_t thread);
```

When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

Searching for a specific pattern in data – the serial version

```
#include     <stdio.h>
#include     <stdlib.h>
#include     <time.h>

//Global variables
const int N=1000000;
int count;

//function prototypes
int count3s(int * , int );

int main()
{
    int i, result;
    int array[N];

//    srand(time(0));
    for (i = 0; i < N; i++)
         array[i] = rand()%100 + 1;
    result = count3s(array, N);
    printf("There were %d 3's\n",result);
    exit(0);
}
```

```
// The serial function to count the number of 3s

int count3s(int * arr, int size)
{
    int i;
    count = 0;
    for (i = 0; i < size; i++)
         if (arr[i] == 3)
            count++;

    return count;
}
```

How to convert a serial code to use threads

- Is there functional or data parallelism? In either case, a function for the thread to execute is needed.
- Is there data that needs to be shared between threads?
- Are the threads sharing any resources?
- Should the threads return a value or values?
- How should the threads terminate?

Choose data parallelism

```
void *count3s_thread(void *id)
{
// Compute the portion of the array that this thread should search
    int myid = *(int *)id, cnt3 = 0, i;
    int length_per_thread = N/tNUM;  //accessing global variables
    int start = myid * length_per_thread;

    for ( i = start; i < start + length_per_thread; i++)
        if(array[i] == 3)
          cnt3++;

    count[myid] = cnt3;    //accessing a global variable

    pthread_exit(NULL);
}
```

Function that creates the threads

```
// This function creates tNUM threads
int count3s(void)
{
    int i;
    int thr_data[tNUM];
    int countt = 0;
    pthread_t thr[tNUM];
    for (i = 0; i < tNUM; i++) {
      thr_data[i] = i;
      pthread_create(&thr[i], NULL, count3s_thread, (void *)&thr_data[i]);
    }

    /* block until all threads complete */
    for (i = 0; i < tNUM; ++i) {
        pthread_join(thr[i], NULL);
        countt += count[i];
    }

    return countt;
}
```

The shared global data

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

//Global variables
const int N=1000000;
const int tNUM = 4;
int array[1000000];
int count[4];
//function prototypes
int count3s(void);
void *count3s_thread(void *);

int main(){
    int i, result;
//    srand(time(0));

    for (i = 0; i < N; i++)
          array[i] = rand()%100 + 1;
    result = count3s();
    printf("There were %d 3's\n",result);
    exit(0);
}
```

Threads and synchronization

The execution of multiple threads in a process often needs to be coordinated:

- Prevent inadvertent access to shared data.
- Prevent a possible race condition, e.g. results need to be written to a shared file in a specific order.
- A task can not be performed until another task has been completed by a different thread.
- A barrier needs to created to wait for all threads to complete a task before proceeding.
- Synchronization functions allow the programmer to control the scheduling and execution of threads

Mutexes

A mutex is a mutual exclusion lock:

- Block access to variables by other threads.
- Enforces exclusive access by a thread to a shared resource.
- Can protect a “critical” section of memory.

Mutexes can be applied to threads in a single process and do not work between processes

Basic mutex functions

Declaring a mutex:

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
```

Using a mutex:

```
    int pthread_mutex_lock(pthread_mutex_t *mutex);
    int pthread_mutex_trylock(pthread_mutex_t *mutex);
    int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Destroying a mutex:

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

Mutex usage – access to a shared variable

Code with no mutex

```
int counter=0;

/* Function C */


void functionC()

{

   counter++

}
```

Code with a mutex

```
/* mutex and counter are global variables */

pthread_mutex_t mutex1 =
PTHREAD_MUTEX_INITIALIZER;

int counter=0;


/* Function C */

void functionC()

{

   pthread_mutex_lock( &mutex1 );

   counter++

   pthread_mutex_unlock( &mutex1 );

}
```

Alternative way to initialize a mutex

A function can be used to initial a mutex and change its properties:

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

If *attr is set to NULL, a default mutex is created.

Advantages and disadvantages of mutexes

Advantages:

- Most efficient way to program mutual exclusion
- Easy to program

Disadvantages:

- Using multiple mutexes can lead to deadlock.
- Only thread that locks the mutex can unlock it.
- Only one thread can execute at a time.

Condition Variables

The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true.

More than one thread can be allowed to execute a critical section of code.

A condition variable must always be associated with a mutex to avoid a race condition created by one thread preparing to wait and another thread which is preparing to change the truth value of the condition.

There is no explicit link between the mutex and the condition variable.

Basic condition variable functions

Declaring a condition variable:

```
pthread_cond_init(pthread_cond_t *cond,
          const pthread_condattr_t *attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Destroying a condition variable:

```
pthread_cond_destroy(pthread_cond_t *cond);
```

Using a condition variable

Waiting on condition:

pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex); - unlocks the mutex and waits for the condition variable cond to be signaled.

pthread_cond_timedwait(pthread_cond_t *cond,
        pthread_mutex_t *mutex,const struct timespec *abstime); - place limit on how long it will block.

Waking thread based on condition:

pthread_cond_signal(pthread_cond_t *cond); - restarts one of the threads that are waiting on the condition variable cond.

pthread_cond_broadcast(pthread_cond_t *cond); - wake up all threads blocked by the specified condition variable.

Interaction of condition variables in threads

Thread X

1) Acquires mutex A
2) Checks condition (T or F)
3) False – mutex A released atomically & thread X suspended
4) …
5) Thread reaquires mutex A (repeat 1) if false)
6) True – proceed with execution of critical section
7) Thread finishes critical section and releases mutex A
8) Thread continues its instructions

Thread Y

1) Acquires mutex A
2) It modifies the state of the condition
3) Signal message to unblock X (and any other threads)
4) Thread releases mutex A
5) Thread continues its instructions

Posix semaphores

All POSIX semaphore functions and types are defined in semaphore.h. To define a semaphore object, use

```
#include <semaphore.h>
sem_t sem_name;
```

To initialize a semaphore, use sem_init():

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- sem points to a semaphore object to initialize
- pshared is a flag indicating whether or not the semaphore should be shared with a process related by a common ancestor and using shared memory.
- value is the initial value assigned to the semaphore
-

Example of use:

```
sem_init(&sem_name, 0, 10);
```

The value of pshared is 0, so the semaphore is known only in this process.

Using semaphores

A thread can have exclusive access to a semaphore (binary semaphore) or several threads can be accessing the semaphore (counting semaphore). The programmer decides the usage by the appropriate use of sem_wait() and sem_post().

To access or obtain a semaphore,

```
int sem_wait(sem_t *sem);
```

Example of use:

```
sem_wait(&sem_name);
```

- This function decrements the value of the semaphore.
- If the value of the semaphore is negative, the calling thread blocks until another thread increments the value.

To increment the value of a semaphore, use sem_post():

```
    int sem_post(sem_t *sem);
```

Example of use:

```
    sem_post(&sem_name);
```

The function increments the value of the semaphore and wakes up any blocked thread that is waiting on the semaphore.

Any thread can issue a call to sem_post() and is issued when the thread is done with a critical section of code or no longer needs exclusive access to a shared variable or resource.

To find out the value of a semaphore, use

```
int sem_getvalue(sem_t *sem, int *valp);
```

The current value of sem is placed in the location pointed to by valp.

Example of use:

```
int value;
sem_getvalue(&sem_name, &value);
printf("The value of the semaphore is %d\n", value);
```

To destroy a semaphore, use

```
int sem_destroy(sem_t *sem);
```

No threads should be waiting on the semaphore or it will not be destroyed.

Example of use:

```
sem_destroy(&sem_name);
```

Thread safe functions

A thread function must call other functions which are "thread safe".

- This means that there are no static or global variables which other threads may accidently change.
- If static or global variables are used then mutexes or other synchronization must be used to protect the variables.
- In C, local variables are dynamically allocated on the stack. Any function that only uses local variables and does not use static data or other shared resources is thread-safe.
- Many library functions are thread-safe. POSIX.1-2001 and POSIX.1-2008 require that all functions specified in the standard shall be thread-safe, except for a specified few. See the man pages, man pthreads.

Scheduling of threads

The scheduling of threads can be specified:

during thread creation – set the attribute parameter:

- detached state – Default, PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED
- scheduling policy - PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED, SCHED_OTHER
- scheduling parameter
- inheritsched attribute – Default, PTHREAD_EXPLICIT_SCHED or inherit from parent thread, PTHREAD_INHERIT_SCHED
- Scope - Kernel threads: PTHREAD_SCOPE_SYSTEM. User threads, PTHREAD_SCOPE_PROCESS. Pick one or the other not both.
- guard size
- stack address - _POSIX_THREAD_ATTR_STACKADDR)
- stack size - default minimum PTHREAD_STACK_SIZE set in pthread.h.

Scheduling cont.

Dynamically changing the attributes of a thread already created:

```
int pthread_getschedparam(pthread_t thread, int *policy,
    struct sched_param *param);

int pthread_setschedparam(pthread_t thread, int policy,
    const struct sched_param *param);
```

Defining the effect of a mutex on the thread's scheduling when creating a mutex.

The pthreads library provides default values that are sufficient for most cases.