Minor 1: kernel and user space

==

Need to be demo-ed to Riju 3+3+6+6+6 = 24

==

1. Do a UART loopback test on the PI. Connect the tx and rx pins with a wire and start minicom. Disable local echo. Then type in something in the minicom terminal. If you see what you type, then the tx is sending those characters to the rx pins through local loopback and your uart tx-rx pins are working. Tutorial: https://www.raspberrypi.org/forums/viewtopic.php?t=148440.

2. Write a "hello world (on insmod), goodbye world (on rmmod)" loadable kernel module. Build it (locally compile on PI or cross compile on host) and insmod/rmmod it from Raspbian command line. Show the kernel print messages.

3. Convert 2 GPIO pins on the PI to UART pins using the pigpio library http://abyz.me.uk/rpi/pigpio/. Do the tx-rx loopback as before, but now connecting the two GPIO pins with a wire. Tutorial: https://www.rs-online.com/designspark/raspberry-pi-2nd-uart-a-k-a-bit-banging-a-k-a-software-serial.

4. Convert the same 2 GPIO pins as above with kernel modules. Again test the tx-rx loopback, connecting the two GPIO pins with a wire. The two kernel modules to try:

(a) https://github.com/adrianomarto/soft_uart/,

(b) https://github.com/themrleon/RpiSoft-UART.

==

Need to be uploaded as a text file in Moodle 2+2+2 = 6

==

5. List the system calls that the pigpio library uses, to talk to hardware through the linux kernel. Describe in a line or two what each such function does.

6. List the functions in the two kernel modules that talk to the hardware. Describe in a line or two what each such function does.

7. List the functions in the two kernel modules that user space processes can call. Describe in a line or two what each such function does.

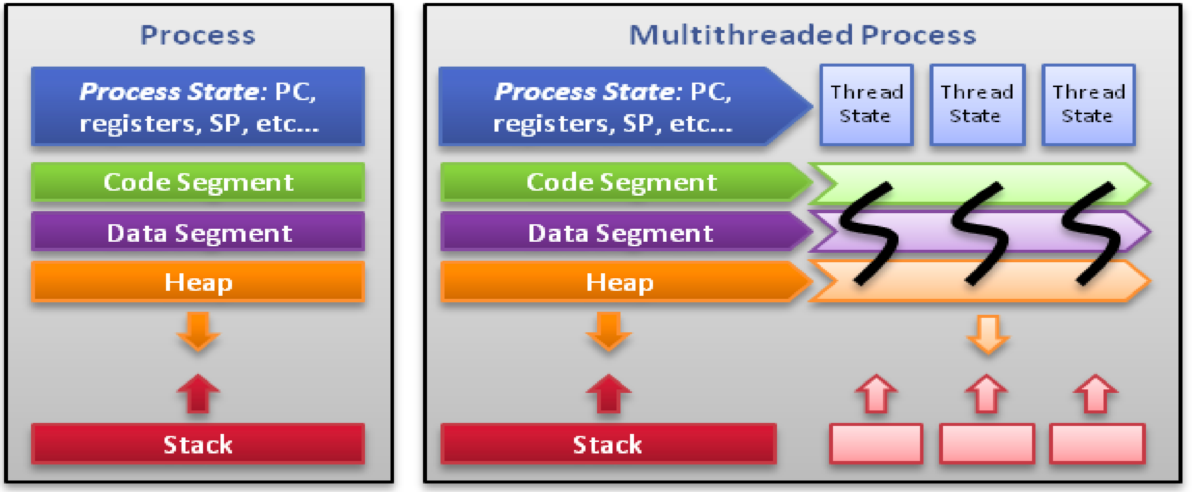
Kernel Threads or Light Weight Processes (LWP): Unit of scheduling in Linux Kernel

Why Light Weight?

- A computer program becomes a process when it is loaded from some store into the computer's memory and begins execution. A process can be executed by a processor or a set of processors. A process description in memory contains vital information such as the program counter which keeps track of the current position in the program (i.e. which instruction is currently being executed), registers, variable stores, file handles, signals, and so forth.

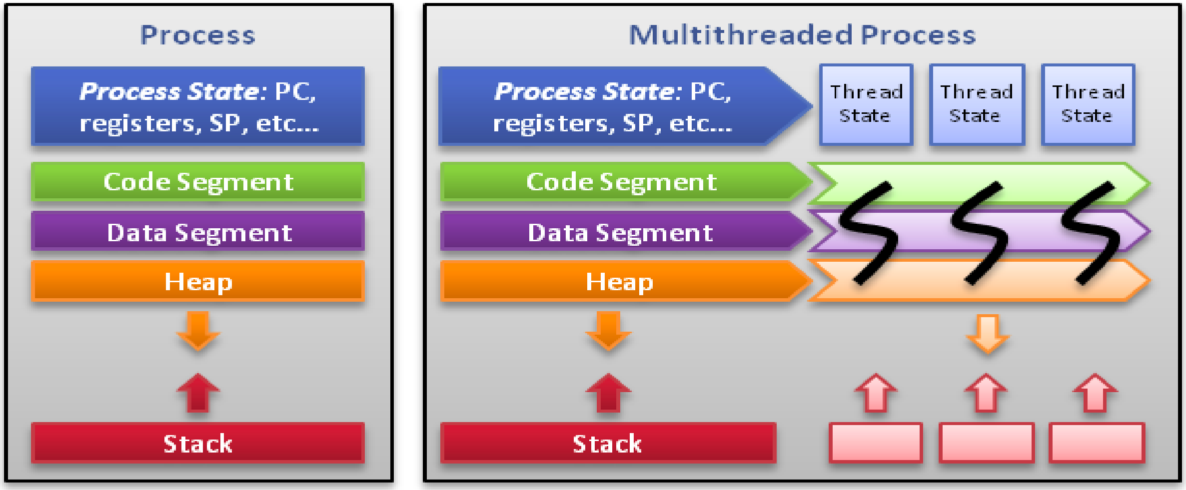
Why Light Weight?

- A computer program becomes a process when it is loaded from some store into the computer's memory and begins execution. A process can be executed by a processor or a set of processors. A process description in memory contains vital information such as the program counter which keeps track of the current position in the program (i.e. which instruction is currently being executed), registers, variable stores, file handles, signals, and so forth.
- A thread is a sequence of such instructions within a program that can be executed independently of other code. Threads are within the same process address space, thus, much of the information present in the memory description of the process can be shared across threads.
- Some information cannot be replicated, such as the stack (stack pointer to a different memory area per thread), registers and thread-specific data. This information sufficies to allow threads to be scheduled independently of the program's main thread and possibly one or more other threads within the program.



Why Light Weight?

- A computer program becomes a process when it is loaded from some store into the computer's memory and begins execution. A process can be executed by a processor or a set of processors. A process description in memory contains vital information such as the program counter which keeps track of the current position in the program (i.e. which instruction is currently being executed), registers, variable stores, file handles, signals, and so forth.
- A thread is a sequence of such instructions within a program that can be executed independently of other code. Threads are within the same process address space, thus, much of the information present in the memory description of the process can be shared across threads.
- Some information cannot be replicated, such as the stack (stack pointer to a different memory area per thread), registers and thread-specific data. This information sufficies to allow threads to be scheduled independently of the program's main thread and possibly one or more other threads within the program.
- Explicit operating system support is required to run multithreaded programs. Fortunately, most modern operating systems support threads such as Linux (via NPTL), BSD variants, Mac OS X, Windows, Solaris, AIX, HP-UX, etc. Operating systems may use different mechanisms to implement multithreading support.



POSIX Threads or pthreads

- Standard that unix vendors have to support, to enable creating threads in userspace

POSIX Threads or pthreads

- Standard that unix vendors have to support, to enable creating threads in userspace
- LINUX gives userspace library Native POSIX Threads Library (NPTL) as part of glibc

```
rijurekha@rijurekha-Inspiron-5567:~/Downloads/acmart-master$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.19
```

POSIX Threads or pthreads

- Standard that unix vendors have to support, to enable creating threads in userspace
- LINUX gives userspace library Native POSIX Threads Library (NPTL) as part of glibc
- We can write C code that includes the pthread.h library and call the supported functions
- We also write higher level code in the user space e.g. in Python, C interpreters of which internally includes pthread.h library

```
rijurekha@rijurekha-Inspiron-5567:~/Downloads/acmart-master$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.19
```

Mapping from userspace threads to kernel threads/ Light Weight Processes (LWP)

Relationship between user-level and kernel-level threads

– 1:1

– N:1

– M:N

Mapping from userspace threads to kernel threads/ Light Weight Processes (LWP)

Relationship between user-level and kernel-level threads

– 1:1

each user-level thread maps to one kernel-level thread

e.g. win32, LinuxThreads (1996), Linux NPTL, windows 7, FreeBSD

– N:1

purely user-level threads, kernel is not aware of the existence of threads

e.g. Early version of Java, Solaris Green Thread

– M:N

In 2003, IBM released the Next Generation POSIX Threads (NGPT), which offered substantial improvements over LinuxThreads. It improved support for the POSIX standard, and was notable for providing an M:N threading model in which M user-space threads are executed on N kernel threads. Not used in linux, NPTL became mainstream and not NGPT.

http://www.drdobbs.com/open-source/nptl-the-new-implementation-of-threads-f/184406204

Pthread APIs

- Thread management
- Thread synchronization

More than 100 subroutines

Create API

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

void *print_message_function( void *ptr );

main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}

void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

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

-
- thread - returns the thread id. (unsigned long int defined in pthreadtypes.h)
- attr - Set to NULL if default thread attributes are used. (else define members of the struct pthread_attr_t defined in pthreadtypes.h). Attributes include:
  - detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
  - scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
  - scheduling parameter
  - inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
  - scope (PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS)
  - guard size
  - stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
  - stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h),
- void * (*start_routine) - pointer to the function to be threaded. Function has a single argument: pointer to void.
- *arg - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

```c
void pthread_exit(void *retval);
```

- This routine kills the thread. The pthread_exit function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using pthread_join.
- Note: the return pointer *retval, must not be of local scope otherwise it would cease to exist once the thread terminates.

Change stack size

```c
#include <pthread.h>
#include <stdio.h>
#define NTHREADS 4
#define N 1000
#define MEGEXTRA 1000000

pthread_attr_t attr;

void *dowork(void *threadid)
{
   double A[N][N];
   int i,j;
   long tid;
   size_t mystacksize;

   tid = (long)threadid;
   pthread_attr_getstacksize (&attr, &mystacksize);
   printf("Thread %ld: stack size = %li bytes \n", tid, mystacksize);
   for (i=0; i<N; i++)
     for (j=0; j<N; j++)
      A[i][j] = ((i*j)/3.452) + (N-i);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NTHREADS];
   size_t stacksize;
   int rc;
   long t;

   pthread_attr_init(&attr);
   pthread_attr_getstacksize (&attr, &stacksize);
   printf("Default stack size = %li\n", stacksize);
   stacksize = sizeof(double)*N*N+MEGEXTRA;
   printf("Amount of stack needed per thread = %li\n",stacksize);
   pthread_attr_setstacksize (&attr, stacksize);
   printf("Creating threads with stack size = %li bytes\n",stacksize);
   for(t=0; t<NTHREADS; t++){
      rc = pthread_create(&threads[t], &attr, dowork, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   printf("Created %ld threads.\n", t);
   pthread_exit(NULL);
}
```

Mutex API

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

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;

   /* Create independent threads each of which will execute functionC */

   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc1);
   }

   if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc2);
   }

   /* Wait till threads are complete before main continues. Unless we  */
   /* wait we run the risk of executing an exit which will terminate   */
   /* the process and all threads before the threads have completed.   */

   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(0);
}

void *functionC()
{
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

Compile and run

Compile:

C compiler: cc -lpthread pthread1.c

or

C++ compiler: g++ -lpthread pthread1.c

Run:

./a.out

Results:

Thread 1

Thread 2

Thread 1 returns: 0

Thread 2 returns: 0

Compile:

cc -lpthread mutex1.c

Run:

./a.out

Results:

Counter value: 1

Counter value: 2

Join API

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

#define NTHREADS 10
void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   pthread_t thread_id[NTHREADS];
   int i, j;

   for(i=0; i < NTHREADS; i++)
   {
      pthread_create( &thread_id[i], NULL, thread_function, NULL );
   }

   for(j=0; j < NTHREADS; j++)
   {
      pthread_join( thread_id[j], NULL);
   }

   /* Now that all threads are complete I can print the final result.     */
   /* Without the join I could be printing a value before all the threads */
   /* have been completed.                                                */

   printf("Final counter value: %d\n", counter);
}

void *thread_function(void *dummyPtr)
{
   printf("Thread number %ld\n", pthread_self());
   pthread_mutex_lock( &mutex1 );
   counter++;
   pthread_mutex_unlock( &mutex1 );
}
```

https://code.woboq.org/userspace/glibc/nptl/

```
pthread_attr_setinheritsched.c
pthread_attr_setschedparam.c
pthread_attr_setschedpolicy.c
pthread_attr_setscope.c
pthread_attr_setstack.c
pthread_attr_setstackaddr.c
pthread_attr_setstacksize.c
pthread_barrier_destroy.c
pthread_barrier_init.c
pthread_barrier_wait.c
pthread_barrierattr_destroy.c
pthread_barrierattr_getpshared.c
pthread_barrierattr_init.c
pthread_barrierattr_setpshared.c
pthread_cancel.c
pthread_clock_gettime.c
pthread_clock_settime.c
pthread_cond_broadcast.c
pthread_cond_common.c
pthread_cond_destroy.c
pthread_cond_init.c
pthread_cond_signal.c
pthread_cond_wait.c  _condvar_cleanup_buffer
pthread_condattr_destroy.c
pthread_condattr_getclock.c
pthread_condattr_getpshared.c
pthread_condattr_init.c
pthread_condattr_setclock.c
pthread_condattr_setpshared.c
pthread_create.c
pthread_detach.c
pthread_equal.c
pthread_exit.c
pthread_getaffinity.c
```

https://code.woboq.org/userspace/glibc/nptl/pthread_create.c.html

```
int
__pthread_create_2_1 (pthread_t *newthread, const pthread_attr_t *attr,
                      void *(*start_routine) (void *), void *arg)
{
  STACK_VARIABLES;

  const struct pthread_attr *iattr = (struct pthread_attr *) attr;
  struct pthread_attr default_attr;
  bool free_cpuset = false;
  if (iattr == NULL)
    {
      lll_lock (__default_pthread_attr_lock, LLL_PRIVATE);
      default_attr = __default_pthread_attr;
      size_t cpusetsize = default_attr.cpusetsize;
      if (cpusetsize > 0)
        {
          cpu_set_t *cpuset;
          if (__glibc_likely (__libc_use_alloca (cpusetsize)))
            cpuset = __alloca (cpusetsize);
          else
            {
              cpuset = malloc (cpusetsize);
              if (cpuset == NULL)
                {
                  lll_unlock (__default_pthread_attr_lock, LLL_PRIVATE);
                  return ENOMEM;
```

User space code in Python

```python
# Python program to illustrate the concept
# of threading
import threading
import os

def task1():
    print("Task 1 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 1: {}".format(os.getpid()))

def task2():
    print("Task 2 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 2: {}".format(os.getpid()))

if __name__ == "__main__":

    # print ID of current process
    print("ID of process running main program: {}".format(os.getpid()))

    # print name of main thread
    print("Main thread name: {}".format(threading.main_thread().name))

    # creating threads
    t1 = threading.Thread(target=task1, name='t1')
    t2 = threading.Thread(target=task2, name='t2')

    # starting threads
    t1.start()
    t2.start()

    # wait until all threads finish
    t1.join()
    t2.join()
```

Run on IDE

```
ID of process running main program: 11758
Main thread name: MainThread
Task 1 assigned to thread: t1
ID of process running task 1: 11758
Task 2 assigned to thread: t2
ID of process running task 2: 11758
```

Python Interpreter internally calls pthread APIs

https://github.com/enthought/Python-2.7.3/blob/master/Python/thread_pthread.h
https://github.com/python/cpython/blob/master/Python/thread.c

Branch: master ▾ **Python-2.7.3 / Python / thread_pthread.h** Find file Copy path

cournape Python 2.7.3. 69fe0ff on 21 Dec 2013

1 contributor

506 lines (420 sloc) | 13 KB Raw Blame History

```c

/* Posix threads interface */

#include <stdlib.h>
#include <string.h>
#if defined(__APPLE__) || defined(HAVE_PTHREAD_DESTRUCTOR)
#define destructor xxdestructor
#endif
#include <pthread.h>
```

```c
/* for safety, ensure a viable minimum stacksize */
#define THREAD_STACK_MIN        0x8000  /* 32kB */
```

```c
long
PyThread_start_new_thread(void (*func)(void *), void *arg)
{
    pthread_t th;
    int status;
#if defined(THREAD_STACK_SIZE) || defined(PTHREAD_SYSTEM_SCHED_SUPPORTED)
    pthread_attr_t attrs;
#endif
#if defined(THREAD_STACK_SIZE)
    size_t      tss;
#endif

    dprintf(("PyThread_start_new_thread called\n"));
    if (!initialized)
        PyThread_init_thread();

#if defined(THREAD_STACK_SIZE) || defined(PTHREAD_SYSTEM_SCHED_SUPPORTED)
    if (pthread_attr_init(&attrs) != 0)
        return -1;
#endif
#if defined(THREAD_STACK_SIZE)
    tss = (_pythread_stacksize != 0) ? _pythread_stacksize
                                     : THREAD_STACK_SIZE;
    if (tss != 0) {
        if (pthread_attr_setstacksize(&attrs, tss) != 0) {
            pthread_attr_destroy(&attrs);
            return -1;
        }
    }
#endif
#if defined(PTHREAD_SYSTEM_SCHED_SUPPORTED)
    pthread_attr_setscope(&attrs, PTHREAD_SCOPE_SYSTEM);
#endif

    status = pthread_create(&th,
#if defined(THREAD_STACK_SIZE) || defined(PTHREAD_SYSTEM_SCHED_SUPPORTED)
                             &attrs,
#else
                             (pthread_attr_t*)NULL,
#endif
                             (void* (*)(void *))func,
                             (void *)arg
                             );
```

User space code in Java,
JVM internally calls pthread APIs

```java

public class Thread {
   static AtomicInteger threadCount = new AtomicInteger(1);

   public void run() {
     System.out.println("Running Thread " + threadCount.getAndIncrement());
   }

   public void start() {
       start0();
   }
   private native void start0();
}
```

```cpp
JNIEXPORT void JNICALL Java_com_threading_Thread_start0(JNIEnv *env, jobject javaThreadObjectRef)
{
    //Get jvm instance and global reference to Thread java object to be passed to
    //pthread entry point function.
    JavaThreadWrapper* args = new JavaThreadWrapper(env, javaThreadObjectRef);

    //init thread attributes
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    //native thread id
    pthread_t tid;
    if (pthread_create(&tid, &attr, thread_entry_point, args))
    {
        fprintf(stderr, "Error creating thread\n");
        return;
    }

    std::cout << "Started a linux thread " << tid << "!" << endl;
    return;
}
```

What if the python interpreter or the JVM does something weird?

What if the python interpreter or the JVM does something weird?

http://www.dabeaz.com/python/GIL.pdf

A Performance Experiment

- Consider this trivial CPU-bound function

```
def count(n):
    while n > 0:
        n -= 1
```

- Run it twice in series

```
count(100000000)
count(100000000)
```

- Now, run it in parallel in two threads

```
t1 = Thread(target=count,args=(100000000,))
t1.start()
t2 = Thread(target=count,args=(100000000,))
t2.start()
t1.join(); t2.join()
```

What if the python interpreter or the JVM does something weird?

http://www.dabeaz.com/python/GIL.pdf

A Performance Experiment

- Consider this trivial CPU-bound function

```
def count(n):
    while n > 0:
        n -= 1
```

- Run it twice in series

```
count(100000000)
count(100000000)
```

- Now, run it in parallel in two threads

```
t1 = Thread(target=count,args=(100000000,))
t1.start()
t2 = Thread(target=count,args=(100000000,))
t2.start()
t1.join(); t2.join()
```

A Mystery

- Why do I get these performance results on my Dual-Core MacBook?

Sequential : 24.6s
Threaded : 45.5s (1.8X slower!)

- And if I disable one of the CPU cores, why does the threaded performance get better?

Threaded : 38.0s