Lecture VI
Threads

§1. Introduction

- While Threads is an important topic on its own right, we have a different reason for threads in this book: we want to explore issues of processes by simulate them as threads at the user-level!
- The best way to understand threads is to think of them as mini-processes. A process can have one or more threads.
- Motivation:

  Your program must break up its task into work that can be done by independent concurrent threads.
- EXAMPLE: Web server application.

  Dispatcher thread: receive incoming requests

  Display thread: obvious

  Network thread: retrieving data from network,

  Worker threads: to handle each request...

  CONSIDER THE ALTERNATIVE (nonthreaded solution): need to carry out each request to completion (but we might be waiting on a disk access, wasting time).
- Benefits of Threads:
  1. Responsiveness (no unnecessary waits)
  2. Resource sharing (threads share common data)
  3. Light weight processes (cheaper than processes for scheduling and on system resources)
  4. Ease of writing code
  5. Ease of exploiting multiprocess architectures
- MODELS of threaded programs.

  (1) Manager/worker: manager thread assigns work to other worker threads. Typically, manager handles all inputs and parcels out work to workers. (can have static or dynamic worker pool)

  (2) Pipeline: like an automobile assembly line.

  (3) Peer: the manager thread is like a worker thread after the initial setup.
- Differences and Similarities:

1. Each thread is an executional entity, like processes. The name "thread of execution" suggests this very well.
2. As such, thread has a PC, a register set, a stack.
3. Unlike processes, all threads in the same process shares the same code section, and shares the data section.
4. Other resources like files, signals, etc, are also shared process-wide.
5. Processes are kernel entities. Some threads can be user entities.

§2. DEADLOCKS (contd)

- Execution Trajectories: Figure 3-8 [p.176] of text.
- Banker's Algorithm.

  SIMPLE EXAMPLE (1 resource)

	Has	Max
A	0	6
B	0	5
C	0	5
D	0	7
Free	10	

	Has	Max
A	1	6
B	1	5
C	2	5
D	4	7
Free	2	

	Has	Max
A	1	6
B	2	5
C	2	5
D	4	7
Free	1	

  Initially, Safe, Unsafe.

- Generalize to multiple resources.

§3. POSIX THREADS

The book's treatment of threads in chapter 2 is somewhat confusing because it goes into issues of threading before clearly explaining what it is. To gain a better understanding of this important concept, we are going to look at **POSIX threads** (or "pthreads"). To ensure portability of code, you should try to program only pthreads. Note: POSIX=portable operation system interface (a registered trademark of IEEE)

- Is pthreads available in cygwin?

  A: Yes! The library is found in C:/cygwin/lib/libpthread.a, and the include file in C:/cygwin/usr/include/pthread.h.

- Tutorials? A: There are various tutorials to be found on the web. One such tutorial (which we henceforth call "pthreadTutor") may be found at Lawrence Livermore National Lab (LLNL), at the URL `www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html`

- Books? A: See "Threads Primer: A guide to multithreaded programming" by Bil Lewis and Daniel J. Berg, Sunsoft Press, 1996.
- What are some models for writing threaded programs?

  Your program must break up its task into work that can be done by independent concurrent threads.

  (1) Manager/worker: manager thread assigns work to other worker threads. Typically, manager handles all inputs and parcels out work to workers. (can have static or dynamic worker pool) [Tanenbaum] illustrates this with web server application.

  (2) Pipeline: line an automobile assembly line.

  (3) Peer: the manager thread is like a worker thread after the initial setup.
- How can I use pthread?

  It is only defined for the C language.

  Your C program must include "pthread.h".

  Please make sure that your `#include <pthread.h>` statement is the very first of all your include files. REASON: other include files may have "thread-safe" features which are turned on only after pthread.h has been included.

  When you compile and link, be sure to tell your linker to use the pthread library. E.g.,

  `gcc -lpthread mythreadprog.c`
- What is the pthread API?

  There are three main types of objects: threads, mutex variables, condition variables.

  But associated with each main type of objects are corresponding **attribute objects**. Thus we have objects for (resp.) thread attributes, mutex attributes and condition attributes. Some people call them **opaque objects**.

  The API has over 60 subroutines divided into 3 classes of calls:

  (1) Thread management: create, detach, join, etc. set/query thread attributes (joinable, scheduling, etc).

  (2) Mutexes: create, destroy, lock, unlocking mutexes. set/query mutex attributes.

  (3) Condition variables: communication between threads that share a mutex. Create/destroy/wait/signal. Also set/query condition variable attributes.
- Names in the pthread module can be bewildering at first, so remember the following name prefix conventions:

```
pthread_... are threads themselves and misc.subroutines
pthread_attr_ are thread attribute objects
pthread_mutex_ are mutexes
pthread_mutex_attr are mutex attribute objects
pthread_cond_ are condition vars
pthread_cond_attr are condition attribute objects
pthread_key_ are thread-specific data keys
```

In other words, every method in this API will begin with the prefix `pthread_`.

- Attribute Objects. This is an interesting concept that must be mastered if you want to use pthreads. We focus on thread attributes at first.

  A thread attribute object stores the following information (the default values are in parenthesis):
  Scope (PTHREAD_SCOPE_PROCESS)
  Detached State (PTHREAD_CREATE_JOINABLE)
  Stack Address (NULL)
  Stack Size (NULL)
  Priority (NULL)

  When we create a thread, one of the arguments is an attribute object. The attributes are thereby transferred to the thread.

- HOW TO USE ATTRIBUTE OBJECTS:

  1. You must create an attribute object before using it. Do:

     ```
     // declare attr to be attribute variable
     pthread_attr_t   attr;

     // initialize attr
     if (0 != phtread_attr_init(pthread_attr_t *attr))
          perror("initializing pthread attribute");
     ```

     This returns a 0 if successful, else an error number is returned. Now `attr` is an initial object with all the default values.

  2. You can subsequently change the attribute fields. E.g., to set the detached state, do

     `phtread_attr_setdetachstate(pthread_attr_t *attr, int state)`

     where `state=PTHREAD_CREATE_DETACHED` or `state=PTHREAD_CREATE_JOINABLE`.

  3. To see the current value of the detached state, use

     `phtread_attr_getdetachstate(const pthread_attr_t *attr, int *state)`

  4. If you subsequently change the attribute object, this has no effect on the thread which was created earlier with this object.

5. REMARKS ON STACK: The default is no stack – the stack address and stack size pointers are both NULL in the pthread attribute object.

   When do you need a thread-specific stack? If the thread code is run by more than one thread, you should consider giving your thread its own stack! *BE SURE THAT EACH THREAD HAS A DIFFERENT STACK ADDRESS.*

   To get the thread-specific stack you need to set the stack size and stack address attributes.

6. SETTING STACK SIZE: You might think that this is easy. But there is a catch: if you set it smaller than some constant `PTHREAD_STACK_MIN`, there will be an error (presumably the default stack size of 0 will be used). Also, I found that `PTHREAD_STACK_MIN`, was not defined in my environment. So I had to define my own value and verify that it is big enough. The following seems good:

```
#define PTHREAD_STACK_MIN 10000
size_t my_stacksize;

if (0!= pthread_attr_setstacksize(&attr, (size_t)PTHREAD_STACK_MIN))
     perror("set stack size");
pthread_attr_getstacksize(&attr, &my_stacksize);
        printf("my stack size = %d \n", (int)my_stacksize);
```

7. SETTING A STACK ADDRESS: Well, you first need to allocate the space with the standard system call "malloc".

```
void * stackaddr;               // stack address pointer
stackaddr = (void *)malloc((size_t)PTHREAD_STACK_MIN);

pthread_attr_setstackaddr(&attr, &my_stacksize);
```

8. SCOPE NOTE: The scope of the thread can be "process level" (default) or "system level". A process-scope thread is not visible to the kernel, and a system-scope is. E.g., the system does not schedule a process-level thread.

We also say the process is **unbound** when it has has process-scope, and **bound** otherwise. This terminology arise from the fact that bound processes are identified with a LWP (**light weight process**) which are kernel objects capabable of being scheduled.

- OK, I am ready, how do I create a pthread?

  Initially, there is a single **main thread** associated with the main() program. All other threads must be explicitly created by the programmer using: Here is the exact prototype for the call:

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

Suppose we call: `pthread_create(&thread, &attr, start_routine, (void *)&arg)`.

A successful call returns a 0. The new thread ID is pointed to by the `thread` argument. The caller can use this ID to perform other operations.

The `attr` parameter is used to set thread attributes. You can use a "thread attribute object" or NULL to accept the defaults.

The `start_routine` is a C routine that the thread will execute when it is created. The argument to `start_routine` is a pointer to a void; this argument is the last argument of `pthread_create()`.

The last `arg` is passed to the `start_routine` by reference (recase as a pointer of type void).

- HOW TO TERMINATE THREADS?

  This can happen in several ways:

  – the main (initial) thread returns

  – the thread calls `pthread_exit`

  – it is canceled by another thread via `pthread_cancel`

  – the process is terminated by a call to exit or exec

  Note that the thread_function turns a `void *`. In particular, the routine can call `int pthread_exit(void *status)` to exit, so status contains the return value.

```
int pthread\_exit(void *status)
```

  If the main thread finishes using `pthread_exit()` before its descendent threads, the other threads continue running; if `pthread_exit()` is not used, the descendents are terminated automatically.

  The termination "status" is stored as a void pointer which any thread may join. To understand this remark, we must know that every thread will be defined to have either the DETACHED or the JOINABLE state (but not both). If DETACHED, all its resources and exit status are discarded when the thread terminates. If JOINABLE, this information will be retained until it is "joined" to another thread (and thereby available to this thread). The default state is JOINABLE.

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

CLEANUP: `pthread_exit()` does not close files...

SUGGESTION: Use `pthread_exit()` to exit from all threads, esp. main().

- How about a sample code?

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

void *PrintHello(void *threadid) {
   printf("\n%d: hello world!\n", threadid);
   pthread_exit(NULL);
}

int main (int argc, char * argv[]) {
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for (t=0; t<NUM_THREADS; t++) {
      printf("Creating thread %d\n", t);
      rc = pthread_create( &threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code of pthread_create() is %d\n", rc);
 exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

- Mutexes. Let us consider next the mutex synchronization objects. Each mutex is essentially an int variable that are access in a very stylized way: either **lock** or **unlock**. Let L and U be the number of lock and unlock operations on a mutex variable. If $L > U$, the mutex is in the "lock state", and exactly $L - U$ of the processes that performed the lock operation will be blocked. Any "unlock" operation will unblock one of these processes.

```
int
pthread_mutex_init ( pthread_mutex_t *mut, const pthread_mutexattr_t *attr);
    -- creates a mutex variable
int
pthread_mutex_lock ( pthread_mutex_t *mut);
    -- locks a mutex variable
int
pthread_mutex_unlock ( pthread_mutex_t *mut);
    -- unlocks a mutex variable
int
```

```
pthread_mutex_trylock ( pthread_mutex_t *mut);
    -- either acquires a mutex variable or returns EBUSY.
int
pthread_mutex_destroy ( pthread_mutex_t *mut);
```

- Condition Variables. Condition variables allows you to wait for some predicate to hold on the condition variables.

  It is always used in conjunction with an associated mutex variable.

  There are two main operations on a cond.variable: wait and signal.

  USAGE: this usage is very stylized (in some sense, the concept of cond.variables is not a primitive concept). Let "cond" and "mut" be the condition var. and associated mutex var.
  – HOW TO WAIT: thread locks "mut" then then waits on "cond" AND "mut". When wait returns, it should unlock "mut". [REMARK: while waiting, "mut" is actually unlocked by the wait call; it is locked again upon return from wait call.]
  – HOW TO SIGNAL: thread locks "mut" and then signals "cond". When done, it unlocks "mut".

```
1. Initialisation:
int
pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *attr);

-- as usual, *attr can be NULL.
2. Waiting:
int
pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mut);
-- caller thread always blocks.
-- NOTE that a mutex variable (mut) is required.
-- You MUST lock "mut" before this call, and unlock "mut" after.
-- This call is equivalent to:
a. pthread_mutex_unlock (mut);
b. block_on_cond (cond);
c. pthread_mutex_lock (mut);
-- Note that c. requires the mutex lock to be acquired.
3.  Signalling:
int
pthread_cond_signal (pthread_cond_t *cond);
-- this wakes up (at least?) one thread waiting on cond.
["At least" seems wrong: I think it should be "at most".
E.g., if there are no threads waiting on cond., then
nothing is woken up, according to description of signal.
If there are many, then I think you want to wake
up only one.  OTHERWISE you should use "broadcast"
instead of "signal".]
```

```
-- because of step c. above, the threads will exit
the block, one at a time.
-- You MUST lock "mut" before this call, and unlock "mut" after.
4.  Broadcast Signalling:
int
pthread_cond_broadcast (pthread_cond_t *cond);
-- wakes up all threads blocked on cond.
5.  Waiting with timeout:
int
pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mut,
                        const struct timespec *abstime);
-- similar to wait, but with timeout (absolute time of day)
6.  Deallocation:
int
pthread_cond_destroy (pthread_cond_t *cond);
```

- Example of Use of Conditional Variables

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;

decrement_count()
   { pthread_mutex_lock(&count_lock);

     while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
     count = count - 1;
     pthread_mutex_unlock(&count_lock);
   }

increment_count()
   { pthread_mutex_lock(&count_lock);
     if (count == 0)
       pthread_cond_signal(&count_nonzero);
     count = count + 1;
     pthread_mutex_unlock(&count_lock);
    }
```

Source: David Marshall (www.cs.cf.ac.uk/Dave/C/node31.html)

- Other useful functions:

`pthread_t pthread_self()` returns the its own thread id.

§4. UNIX PIPES

This topic is necessary for doing homework 2.

In hw2, we must solve the problem of computing several GCD's using one master process and many child processes (one per GCD). Only the master process knows how to compute the remainder (modulo operation). Child processes must call the master process to do this computation. We intend to do this communication using pipes.

- Two unix processes can communicate via a pipe.
- A pipe has API like a file. Hence let us discuss files first.
- Actually, ALL UNIX I/O has a file-like API.
- Files that are currently open has an entry into a process table called the FILE DESCRIPTOR TABLE (FDT). An index into this table is called a FILE DESCRIPTOR (fd).
- Every process begins with three default opened files: stdin (fd=0), stdout (fd=1), stderr (fd=2).
- We can redefine these standard files: e.g.

```
% wc < main.c > stats.out
```

- Here is how you replace the standout output (fd=1) by a file named "foo".

```
int fid;
fid = open("foo", O_WRONLY | O_CREAT);     // probably fid=3!
close(1);         // close current stdout.
dup(fid);         // first available entry in FDT is assigned
close(fid);       // can close the this entry in FDT
```

  You are guaranteed to have redirected the process output to the file "foo".

- PIPES: here is how you open a pipe. It is basically a pair of file discriptors!

```
int pipeId [2];   // array of size two
...
pipe (pipeId);    // just created a pipe!
```

  HERE, pipeId[0] is the reading-end of the pipe and pipeId[1] is the writing-end of the pipe.

- HOW TO USE PIPES?
- Some useful techniques in using pipes:
  - To write a sequence of numbers into a pipe, and to read them back as numbers, use sprintf(buf,"then write into the pipe. The receiver reads from the pipe into its own buf, and then uses sscan(buf,"read it back.

– You can set the control bits of a pipe to do a non-blocking read (so you will not block if there is nothing to read).

§5. PThread Example

The models of threading is very diverse, depending on the programming language and the underlying OS.

There are two main models of threads that we will use in our exercises: Posix Threads (PTHREADS) and Java Threads.

The following PTHREAD example from the book illustrates a toy problem involving pipes: the idea is that the main process creates a thread to compute a number. Then the main process waits (join) for the thread to finish. Then the main process finishes.

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

int sum; // global data
void *runner(void *param) //thread

int main(int argc, char *argv[])
{
  pthread_t tid;
  pthread_attr_t attr;

  if (argc != 2) {
   fprintf(stderr, "usage: a.out <integer value>\n");
exit();
  }
  if (atoi(argv[1]) < 0) {
   fprintf(stderr, "argument (%d) must be positive\n",
atoi(argv[1]));
exit();
  }

  // INITIALIZE
  pthread_attr_init(&attr);
  pthread_create(&tid, &attr,runner, argv[1]);

  // WAIT
  pthread_join(tid,NULL);
  printf("sum = %d\n", sum);
}

/* runner thread */
void *runner(void *param)
{
  int i, upper=atoi(param);
  sum=0;

  if (upper >0)
     for (i=1; i<= upper; i++) sum +=i;

  pthread_exit(0);
        }
```

§6. Java Thread Example

§7. Timing in C Programs

You will need to use this information for hw2.

There are two units for time in C and unix: in seconds, and in clock ticks. Clock ticks has a much finer granularity than seconds. The corresponding functions to obtain such times are called `time` and `clock`. Both functions return some integer type (int, long, etc) with the following interpretations:

`time()` returns the number of seconds since Jan 1st 1970 (conventional birth moment of UNIX). ¡li¿ `clock()` returns the number of clock ticks since the execution of the program started. You may identify the clock ticks with the number of CPU cycles. To convert clock ticks into seconds, you must divide by `CLOCKS_PER_SEC`.

These functions and definitions are found in include files called time.h and sys/types.h. E.g., on my computer, the later file is in /usr/include/sys/types.h.

The function `time` takes a pointer to `time_t` as an argument and returns a value of `time_t` on exit. In simple usage, the pointer can be set to NULL, so we only use the return value.

```
/* file: mytime.c
 *
 * Operating Systems Class, Fall 2006, Professor Yap.
 *  THIS code is adapted from the website
 * http://beige.ucs.indiana.edu/B673/node104.html by Zdzislaw Meglicki
 */
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <math.h>

      main()
      {
        time_t  t0, t1;
        clock_t c0, c1;

        long count;
        double a, b, c;

        t0 = time(NULL);
        c0 = clock();

        printf ("\tbegin time in sec:          %ld\n", (long) t0);
        printf ("\tbegin clock ticks:           %d\n", (int) c0);
        printf ("\t\tsleep for 5 seconds ... \n");
        sleep(5);

        printf ("\t\tperform some computation ... \n");
        for (count = 1l; count < 10000000l; count++) {
           a = sqrt(count); b = 1.0/a; c = b - a;
        }

        t1 = time(NULL);
        c1 = clock();

        printf ("\tend time in sec:              %ld\n", (long) t1);
        printf ("\tend clock ticks:              %d\n", (int) c1);
        printf ("\telapsed elapsed time in sec:  %ld\n", (long) (t1 - t0));
        printf ("\telapsed elapsed ticks in sec: %f\n",
```

```
        (float) (c1 - c0)/CLOCKS_PER_SEC);
    }
```

Saving this program as time.c, we compile and run it on my CYGWIN:

```
yap@prog[68c] gcc time.c -lm
yap@prog[684] a.exe
using UNIX function time to measure wallclock time ...
using UNIX function clock to measure CPU time ...
        begin time in sec:        1159825726
        begin clock ticks:        61
                sleep for 5 seconds ...
                perform some computation ...
        end time in sec:          1159825738
        end clock ticks:          6374
        elapsed time in sec:      12
        elapsed ticks in sec:     6.313000
yap@prog[685]
```

REMARK: In CYGWIN, we could just call "gcc time.c", but in general, you need to tell the compiler to look search the math library "-lm". You might also try "time a.exe" to use the UNIX time function to compare with mytime. The two results should be consistent!

§8. POSIX THREADS

Threads are quite diverse, and so a generic treatment is inadequate to point out differences between the various versions.

We must go into the details of some important realizations of threads for this purpose. The text book has given you a taste of Java Threads.

Now, we are going to look at another realization, **POSIX threads** (or "pthreads"). To ensure portability of code, you should try to program only pthreads. Note: POSIX = "portable operation system interface" (a registered trademark of IEEE).

- Is pthreads available in cygwin?

  A: Yes! The library is found in C:/cygwin/lib/libpthread.a, and the include file in C:/cygwin/usr/include/pthread.h.

- Tutorials? A: There are various tutorials to be found on the web. One such tutorial (which we henceforth call "pthreadTutor") may be found at Lawrence Livermore National Lab (LLNL), at the URL
  `www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html`

- Books? A: See "Threads Primer: A guide to multithreaded programming" by Bil Lewis and Daniel J. Berg, Sunsoft Press, 1996.

- What are some models for writing threaded programs?

  Your program must break up its task into work that can be done by independent concurrent threads.

  (1) Manager/worker: manager thread assigns work to other worker threads. Typically, manager handles all inputs and parcels out work to workers. (can have static or dynamic worker pool) [Tanenbaum] illustrates this with web server application.

  (2) Pipeline: line an automobile assembly line.

  (3) Peer: the manager thread is like a worker thread after the initial setup.

- How can I use pthread?

  It is only defined for the C language.

  Your C program must include "pthread.h".

  Please make sure that your `#include <pthread.h>` statement is the very first of all your include files. REASON: other include files may have "thread-safe" features which are turned on only after pthread.h has been included.

  When you compile and link, be sure to tell your linker to use the pthread library. E.g.,

  `gcc mythreadprog.c -lpthread`

- How about a thread-version of hello world?

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

void *PrintHello(void *threadid) {
   printf("\n%d: hello world!\n", threadid);
   pthread_exit(NULL);
}

int main (int argc, char * argv[]) {
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for (t=0; t<NUM_THREADS; t++) {
      printf("Creating thread %d\n", t);
      rc = pthread_create( &threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code of pthread_create() is %d\n", rc);
 exit(-1);
      }
   }
   pthread_exit(NULL);
}
#########################################
```

  1. We first define the function PrintHello(void *), which is to be run by each thread. It simply prints "hello world!".

     Such a function is called the **thread function** of the running thread.
  2. The main loop just creates 5 (=NUM_THREADS) threads, and then terminates by calling pthread_exit(). Why not a plain exit? Because this would kill all the other threads as well.

- Let us go over the thread creation call in the above program.

  Initially, there is a single **main thread** associated with the main() program. All other threads must be explicitly created by the programmer using: Here is the exact prototype for the call:

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

  1. Suppose we call:

     `pthread_create(&thread, &attr, start_routine, (void *)& myarg)`.

     A successful call returns a 0. The new thread ID is pointed to by the `thread` argument. The caller can use this ID to perform other operations.
  2. The `attr` parameter is used to set thread attributes. You can use a "thread attribute object" or NULL to accept the defaults.
  3. The `start_routine` is a C routine that the thread will execute when it is created. The argument to `start_routine` is a pointer to a void; this is `myarg`, which is the last argument of `pthread_create()`.
  4. As we just noted, the last argument `myarg` is passed to the `start_routine`. It is passed by reference (but must be recast as a pointer of type void).

- What is the pthread API?

  There are three main types of objects: threads, mutex variables, condition variables.

  But associated with each main type of objects are corresponding **attribute objects**. Thus we have objects for (resp.) thread attributes, mutex attributes and condition attributes. Some people call them **opaque objects**.

  The API has over 60 subroutines divided into 3 classes of calls:

(1) Thread management: create, detach, join, etc. set/query thread attributes (joinable, scheduling, etc).

(2) Mutexes: create, destroy, lock, unlocking mutexes. set/query mutex attributes.

(3) Condition variables: communication between threads that share a mutex. Create/destroy/wait/signal. Also set/query condition variable attributes.

- Names in the pthread module can be bewildering at first, so remember the following name prefix conventions:

```
pthread_... are threads themselves and misc.subroutines
pthread_attr_ are thread attribute objects
pthread_mutex_ are mutexes
pthread_mutex_attr are mutex attribute objects
pthread_cond_ are condition vars
pthread_cond_attr are condition attribute objects
pthread_key_ are thread-specific data keys
```

  In other words, every method in this API will begin with the prefix `pthread_`.

- Attribute Objects. This is an interesting concept that must be mastered if you want to use pthreads. We focus on thread attributes at first.

  Attribute objects makes code easier to read – attributes are collected in such objects and they can be passed around!

  A thread attribute object stores the following information (the default values are given):

attribute	Default Value	Explanation
Scope	PTHREAD_SCOPE_PROCESS	process-wide rather than system-wide scope
Detached State	PTHREAD_CREATE_JOINABLE	exit status and resources retained till joined
Stack Address	NULL	default to the system assigned address
Stack Size	NULL	use the default size
Priority	NULL	inherit from parent thread

  When we create a thread, one of the arguments is an attribute object. The attributes are thereby transferred to the thread.

- HOW TO USE ATTRIBUTE OBJECTS:

  1. You must create an attribute object before using it. Do:

     ```
     // declare attr to be attribute variable
     pthread_attr_t   attr;

     // initialize attr
     if (0 != phtread_attr_init(pthread_attr_t *attr))
          perror("initializing pthread attribute");
     ```

     This returns a 0 if successful, else an error number is returned. Now `attr` is an initial object with all the default values.
  2. You can subsequently change the attribute fields. E.g., to set the detached state, do
     `phtread_attr_setdetachstate(pthread_attr_t *attr, int state)`
     where `state=PTHREAD_CREATE_DETACHED` or `state=PTHREAD_CREATE_JOINABLE`.
  3. To see the current value of the detached state, use
     `phtread_attr_getdetachstate(const pthread_attr_t *attr, int *state)`
  4. If you subsequently change the attribute object, this has no effect on the thread which was created earlier with this object.
  5. REMARKS ON STACK: The default is no stack – the stack address and stack size pointers are both NULL in the pthread attribute object.

     When do you need a thread-specific stack? If the thread code is run by more than one thread, you should consider giving your thread its own stack! *BE SURE THAT EACH THREAD HAS A DIFFERENT STACK ADDRESS.*

     To get the thread-specific stack you need to set the stack size and stack address attributes.

6. SETTING STACK SIZE: You might think that this is easy. But there is a catch: if you set it smaller than some constant PTHREAD_STACK_MIN, there will be an error (presumably the default stack size of 0 will be used). Also, I found that PTHREAD_STACK_MIN, was not defined in my environment. So I had to define my own value and verify that it is big enough. The following seems good:

```
#define PTHREAD_STACK_MIN 10000
size_t my_stacksize;

if (0!= pthread_attr_setstacksize(&attr, (size_t)PTHREAD_STACK_MIN))
     perror("set stack size");
pthread_attr_getstacksize(&attr, &my_stacksize);
        printf("my stack size = %d \n", (int)my_stacksize);
```

7. SETTING A STACK ADDRESS: Well, you first need to allocate the space with the standard system call ”malloc”.

```
void * stackaddr;              // stack address pointer
stackaddr = (void *)malloc((size_t)PTHREAD_STACK_MIN);

pthread_attr_setstackaddr(&attr, &my_stacksize);
```

8. SCOPE NOTE: The scope of the thread can be ”process level” (default) or ”system level”. A process-scope thread is not visible to the kernel, and a system-scope is. E.g., the system does not schedule a process-level thread.

We also say the thread is **unbound** when it has has process-scope, and **bound** otherwise. This terminology arise from the fact that bound processes are identified with a LWP (**light weight process**) which are kernel objects capabable of being scheduled.

- HOW TO TERMINATE THREADS?

  This can happen in several ways:

  – the main (initial) thread returns

  – the thread calls `pthread_exit`

  – it is canceled by another thread via `pthread_cancel`

  – the process is terminated by a call to exit or exec

  Note that the thread_function turns a `void *`. In particular, the routine can call `int pthread_exit(void *status)` to exit, so status contains the return value.

```
int pthread\_exit(void *status)
```

  If the main thread finishes using `pthread_exit()` before its descendent threads, the other threads continue running; if `pthread_exit()` is not used, the descendents are terminated automatically.

  The termination ”status” is stored as a void pointer which any thread may join. To understand this remark, we must know that every thread will be defined to have either the DETACHED or the JOINABLE state. Note: DETACHED = not JOINABLE, and JOINABLE = not DETACHED. If DETACHED, all its resources and exit status are discarded when the thread terminates. If JOINABLE, this information will be retained until it is ”joined” to another thread (and thereby available to this thread). The default state is JOINABLE.

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

  In simple applications, you can just set the second argument of `pthread_join` to NULL. But in general, this argument should receive the return status.

  CLEANUP: `pthread_exit()` does not close files...

- Mutexes. Let us consider next the mutex synchronization objects. Each mutex is essentially an int variable that are access in a very stylized way: either **lock** or **unlock**. Let L and U be the number of lock and unlock operations on a mutex variable. If $L > U$, the mutex is in the “lock state”, and exactly $L - U$ of the processes that performed the lock operation will be blocked. Any ”unlock” operation will unblock one of these processes.

```
int
pthread_mutex_init ( pthread_mutex_t *mut, const pthread_mutexattr_t *attr);
    -- creates a mutex variable
int
pthread_mutex_lock ( pthread_mutex_t *mut);
    -- locks a mutex variable
int
pthread_mutex_unlock ( pthread_mutex_t *mut);
    -- unlocks a mutex variable
int
pthread_mutex_trylock ( pthread_mutex_t *mut);
    -- either acquires a mutex variable or returns EBUSY.
int
pthread_mutex_destroy ( pthread_mutex_t *mut);
```

- Condition Variables. Condition variables allows you to wait for some predicate to hold on the condition variables.

  It is always used in conjunction with an associated mutex variable.

  There are two main operations on a cond.variable: wait and signal.

  USAGE: this usage is very stylized (in some sense, the concept of cond.variables is not a primitive concept). Let ”cond” and ”mut” be the condition var. and associated mutex var.
  – HOW TO WAIT: thread locks ”mut” then then waits on ”cond” AND ”mut”. When wait returns, it should unlock ”mut”. [REMARK: while waiting, ”mut” is actually unlocked by the wait call; it is locked again upon return from wait call.]
  – HOW TO SIGNAL: thread locks ”mut” and then signals ”cond”. When done, it unlocks ”mut”.

```
1. Initialisation:
int
pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *attr);

-- as usual, *attr can be NULL.
2. Waiting:
int
pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mut);
-- caller thread always blocks.
-- NOTE that a mutex variable (mut) is required.
-- You MUST lock "mut" before this call, and unlock "mut" after.
-- This call is equivalent to:
a. pthread_mutex_unlock (mut);
b. block_on_cond (cond);
c. pthread_mutex_lock (mut);
-- Note that c. requires the mutex lock to be acquired.
3.  Signalling:
int
pthread_cond_signal (pthread_cond_t *cond);
-- this wakes up (at least?) one thread waiting on cond.
["At least" seems wrong: I think it should be "at most".
E.g., if there are no threads waiting on cond., then
nothing is woken up, according to description of signal.
If there are many, then I think you want to wake
up only one.  OTHERWISE you should use "broadcast"
instead of "signal".]
-- because of step c. above, the threads will exit
the block, one at a time.
-- You MUST lock "mut" before this call, and unlock "mut" after.
4.  Broadcast Signalling:
int
pthread_cond_broadcast (pthread_cond_t *cond);
-- wakes up all threads blocked on cond.
5.  Waiting with timeout:
int
pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mut,
                        const struct timespec *abstime);
-- similar to wait, but with timeout (absolute time of day)
6.  Deallocation:
```

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

- Example of Use of Conditional Variables

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;

decrement_count()
   { pthread_mutex_lock(&count_lock);

     while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
     count = count - 1;
     pthread_mutex_unlock(&count_lock);
   }

increment_count()
   { pthread_mutex_lock(&count_lock);
     if (count == 0)
       pthread_cond_signal(&count_nonzero);
     count = count + 1;
     pthread_mutex_unlock(&count_lock);
    }
```

Source: David Marshall (www.cs.cf.ac.uk/Dave/C/node31.html)

- Other useful functions:

`pthread_t pthread_self()` returns the its own thread id.

Exercises

Exercise 8.1: What are the differences between kernel threads and user threads. ◇

Exercise 8.2: Describe the two general methods for processes to communicate among themselves in UNIX. Discuss how to do this, and their pros/cons. ◇

Exercise 8.3: How do you establish a 2-way communication link between two child processes (called X and Y) using pipes? Go through the mechanics, but do not describe the C program. Be sure to close any I/O files that are not used. ◇

End Exercises