Chapter 12: Thread Control
## Thread Limits: Portability

<table>
<thead>
<tr>
<th>Limit</th>
<th>FreeBSD 8.0</th>
<th>Linux 3.2.0</th>
<th>Mac OS X 10.6.8</th>
<th>Solaris 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTHREAD_DESTRUCTOR_ITERATIONS</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>no limit</td>
</tr>
<tr>
<td>PTHREAD_KEYS_MAX</td>
<td>256</td>
<td>1,024</td>
<td>512</td>
<td>no limit</td>
</tr>
<tr>
<td>PTHREAD_STACK_MIN</td>
<td>2,048</td>
<td>16,384</td>
<td>8,192</td>
<td>8,192</td>
</tr>
<tr>
<td>PTHREAD_THREADS_MAX</td>
<td>no limit</td>
<td>no limit</td>
<td>no limit</td>
<td>no limit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name of limit</th>
<th>Description</th>
<th>name argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTHREAD_DESTRUCTOR_ITERATIONS</td>
<td>maximum number of times an implementation will try to destroy the thread-specific data when a thread exits (Section 12.6)</td>
<td>_SC_THREAD_DESTRUCTOR_ITERATIONS</td>
</tr>
<tr>
<td>PTHREAD_KEYS_MAX</td>
<td>maximum number of keys that can be created by a process (Section 12.6)</td>
<td>_SC_THREAD_KEYS_MAX</td>
</tr>
<tr>
<td>PTHREAD_STACK_MIN</td>
<td>minimum number of bytes that can be used for a thread’s stack (Section 12.3)</td>
<td>_SC_THREAD_STACK_MIN</td>
</tr>
<tr>
<td>PTHREAD_THREADS_MAX</td>
<td>maximum number of threads that can be created in a process (Section 12.3)</td>
<td>_SC_THREAD_THREADS_MAX</td>
</tr>
</tbody>
</table>

**Figure 12.1** Thread limits and *name* arguments to `sysconf`
Thread Attributes

- Thread attributes control the behavior of threads
  - Attribute objects
    - thread attributes
    - mutex attributes
    - etc...

- Thread control functions
  - Initialization function sets up attributes to default values
  - Destruction function frees attributes
  - Get attribute functions
  - Set attribute functions
Thread Attributes: init and destroy

- `pthread_attr_init()` - initializes `attr` with all the default thread attributes
  - `int pthread_attr_init(pthread_attr_t* attr);`

- `pthread_attr_destroy()` – destroys `attr`
  - `int pthread_attr_destroy(pthread_attr_t* attr);`
Thread Attributes:

- `pthread_attr_setdetachstate()` – sets the detached state of a thread
  - `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);`
  - Two legal values:
    - `PTHREAD_CREATE_DETACHED`
    - `PTHREAD_CREATE_JOINABLE`

- `pthread_attr_getdetachstate()` – gets the current detach-state attribute
  - `int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);`
Create a detached thread

```c
#include "apue.h"
#include <pthread.h>

int
makethread(void (*)(*fn)(void *), void *arg)
{
    int err;
    pthread_t tid;
    pthread_attr_t attr;

    err = pthread_attr_init(&attr);
    if (err != 0)
        return(err);
    err = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (err == 0)
        err = pthread_create(&tid, &attr, fn, arg);
    pthread_attr_destroy(&attr);
    return(err);
}
```

**Figure 12.4** Creating a thread in the detached state
Thread Attributes: getstack, setstack

- We have finite virtual address space
  - Each thread gets a stack in that address space
  - We may want to reduce/increase the default stack size

- pthread_attr_setstack() – set stack location and size
  - int pthread_attr_setstack(pthread_attr_t* attr, void* stackaddr, size_t stacksize)

- pthread_attr_getstack() – get stack location and size
  - int pthread_attr_getstack(const pthread_attr_t* restrict attr, void** restrict stackaddr, size_t* restrict stacksize)
Thread Attributes: getstacksize, setstacksize

- If we want to change the default stack size:
  - `pthread_attr_getstacksize()` – get the size of the thread’s stack
    - `int pthread_attr_getstacksize(const pthread_attr_t *restrict attr, size_t *restrict stacksize);`
  - `pthread_attr_setstacksize()` – set the size of the thread’s stack
    - `int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);`
Thread Attributes: getguardsize, setguardsize

- Memory guards are set up to keep stacks from running into one another, to manipulate the default guard values

- `pthread_attr_getguardsize()` – get current guard size
  - `int pthread_attr_getguardsize(const pthread_attr_t* restrict attr, size_t* restrict guardsize);`

- `pthread_attr_setguardsize()` – set current guard size
  - `int pthread_attr_setguardsize(pthread_attr_t* attr, size_t* restrict guardsize);`
Synchronization Attributes

- Mutex attributes
- Reader-writer lock attributes
- Condition variable attributes
- Barrier attributes
Mutex Attributes: init and destroy

- Represented by pthread-mutexattr_t
- Mutex attributes
  - Process-shared
  - Robust
  - Type
- pthread_mutexattr_init() – initializes mutex to default values
  - int pthread_mutexattr_init(pthread_mutexattr_t *attr);
- pthread_mutexattr_destroy() – destroy mutex attributes
  - pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
Mutex Attributes: process-shared

- Provide more efficient mutex implementation
  - PTHREAD_PROCESS_PRIVATE

- pthread_mutexattr_getpshared() – get process-shared attribute
  - int pthread_mutexattr_getpshared(const pthread_mutexattr_t* restrict attr, int* restrict pshared);

- pthread_mutexattr_setpshared() – set process-shared attribute
  - int pthread_mutexattr_setpshared(pthread_mutexattr_t* attr, int* restrict pshared);
mutex attributes: robust

- robust is for mutexes that are shared among processes
  - PTHREAD_MUTEX_STALLED – no action taken when process terminates holding mutex (default)
  - PTHREAD_MUTEX_ROBUST – make pthread_mutex_lock() return EOWNERDEAD instead of 0

- pthread_mutexattr_getrobust() – get robust attribute
  - int pthread_mutexattr_getrobust(const pthread_mutexattr_t* restrict attr, int* restrict robust);

- pthread_mutexattr_setrobust() – set robust attribute
  - int pthread_mutexattr_setrobust(pthread_mutexattr_t* attr, int* restrict robust);
Thread Attributes: Robust consistency

- If we get an EOWNERDEAD from pthread_mutex_lock() we need to call:
  - pthread_mutex_consistent() – Allows mutex to behave normally after it is unlocked
    - int pthread_mutex_consistent(pthread_mutex_t* mutex);

<table>
<thead>
<tr>
<th>Mutex type</th>
<th>Relock without unlock?</th>
<th>Unlock when not owned?</th>
<th>Unlock when unlocked?</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTHREAD_MUTEX_NORMAL</td>
<td>deadlock</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>PTHREAD_MUTEX_ERRORCHECK</td>
<td>returns error</td>
<td>returns error</td>
<td>returns error</td>
</tr>
<tr>
<td>PTHREAD_MUTEX_RECURSIVE</td>
<td>allowed</td>
<td>returns error</td>
<td>returns error</td>
</tr>
<tr>
<td>PTHREAD_MUTEX_DEFAULT</td>
<td>undefined</td>
<td>undefined</td>
<td>undefined</td>
</tr>
</tbody>
</table>

**Figure 12.5** Mutex type behavior
Thread Attributes: gettype, settype

- Type determines whether it can be locked again by a thread that owns it
  - Default type is “normal”

- `pthread_mutexattr_gettype()` – get mutex type
  - `int pthread_mutexattr_gettype(const pthread_mutexattr_t* restrict attr, int* restrict type);`

- `pthread_mutexattr_settype()` – set mutex type
  - `int pthread_mutexattr_settype(pthread_mutexattr_t* attr, int* type);`
Figure 12.6 Recursive locking opportunity
Figure 12.7  Avoiding a recursive locking opportunity
Reader-Writer Locks Attributes

- `pthread_rwlockattr_init()` – Initialize attribute for reader-writer lock
  - `int pthread_rwlockattr_init(pthread_rwlockattr_t* attr);`

- `pthread_rwlockattr_destroy()` – Destroy attribute for reader-writer lock
  - `int pthread_rwlockattr_destroy(pthread_rwlockattr_t* attr);`
RD/RW Lock Attributes: Process Shared

- `pthread_rwlockattr_getpshared()` – Get process-shared attribute
  - `int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t* restrict attr, int* restrict pshared);`

- `pthread_rwlockattr_setpshared()` – Set process-shared attribute
  - `int pthread_rwlockattr_setpshared(pthread_rwlockattr_t* attr, int* pshared);`
Condition Variables

- **pthread_condattr_init()** – initialize condition variable attribute
  - `int pthread_condattr_init(pthread_condattr_t* attr);`

- **pthread_condattr_destroy()** – destroy condition variable attribute
  - `int pthread_condattr_destroy(pthread_condattr_t* attr);`

- **pthread_condattr_getpshared()** – get process shared attribute of conditional variable
  - `int pthread_condattr_getpshared(const pthread_condattr_t* restrict attr, int* restrict pshared);`

- **pthread_condattr_setpshared()** – set process shared attribute of conditional variable
  - `int pthread_condattr_setpshared(pthread_condattr_t* attr, int pshared);`
Conditional Variables: Clocks

- The clock attribute controls which clock is used for `pthread_cond_timedwait()`

- `pthread_condattr_getclock()` – get clock ID
  ```c
  int pthread_condattr_getclock(const pthread_condattr_t* restrict attr, int* restrict clock_id);
  ```

- `pthread_condattr_setpshared()` – set clock ID
  ```c
  int pthread_condattr_setclock(pthread_condattr_t* attr, int clock_id);
  ```
Barrier Attributes

- pthread_barrierattr_init() – initialize barrier attribute
  - int pthread_barrierattr_init(pthread_barrierattr_t* attr);

- pthread_barrierattr_destroy() – destroy barrier attribute
  - int pthread_barrierattr_destroy(pthread_barrierattr_t* attr);

- pthread_barrierattr_getpshared() – get process shared attribute of barrier
  - int pthread_barrierattr_getpshared(const pthread_barrierattr_t* restrict attr, int* restrict pshared);

- pthread_barrierattr_setpshared() – set process shared attribute of barrier
  - int pthread_barrierattr_setpshared(pthread_barrierattr_t* attr, int pshared);
Reentrant functions

- A reentrant function can be interrupted in the middle of its execution and safely reentered again.
- A function is thread-safe if it can be called by multiple threads at the same time.
- A function is async-signal safe if it is safe to reenter it from an asynchronous signal handler.
### Non Thread-Safe Functions

<table>
<thead>
<tr>
<th>basename</th>
<th>getchar_unlocked</th>
<th>getservent</th>
<th>putc_unlocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>catgets</td>
<td>getdate</td>
<td>getuxtent</td>
<td>putchar_unlocked</td>
</tr>
<tr>
<td>crypt</td>
<td>getenv</td>
<td>getuxtid</td>
<td>putenv</td>
</tr>
<tr>
<td>dbm_clearerr</td>
<td>getgrent</td>
<td>gmtime</td>
<td>putuxtline</td>
</tr>
<tr>
<td>dbm_close</td>
<td>getgrgid</td>
<td>hcreate</td>
<td>rand</td>
</tr>
<tr>
<td>dbm_delete</td>
<td>getgrnam</td>
<td>hdestroy</td>
<td>readdr</td>
</tr>
<tr>
<td>dbm_error</td>
<td>gethostent</td>
<td>hsearch</td>
<td>setenv</td>
</tr>
<tr>
<td>dbm_fetch</td>
<td>getlogin</td>
<td>inet_ntoa</td>
<td>setgrent</td>
</tr>
<tr>
<td>dbm_firstkey</td>
<td>getnetbyaddr</td>
<td>l64a</td>
<td>setkey</td>
</tr>
<tr>
<td>dbm_nextkey</td>
<td>getnetbyname</td>
<td>lgamma</td>
<td>spwent</td>
</tr>
<tr>
<td>dbm_open</td>
<td>getnetent</td>
<td>lgammaf</td>
<td>setutxent</td>
</tr>
<tr>
<td>dbm_store</td>
<td>getopt</td>
<td>lgammal</td>
<td>strerror</td>
</tr>
<tr>
<td>dirname</td>
<td>getprotobyname</td>
<td>localeconv</td>
<td>strsignal</td>
</tr>
<tr>
<td>dleerror</td>
<td>getprotybnumber</td>
<td>localtime</td>
<td>strtok</td>
</tr>
<tr>
<td>drand48</td>
<td>getprotoent</td>
<td>lrand48</td>
<td>system</td>
</tr>
<tr>
<td>encrypt</td>
<td>getpwent</td>
<td>mrand48</td>
<td>tttname</td>
</tr>
<tr>
<td>endgrent</td>
<td>getpwnam</td>
<td>nftw</td>
<td>unsetenv</td>
</tr>
<tr>
<td>endpwent</td>
<td>getpwuid</td>
<td>nftw</td>
<td>wcstombs</td>
</tr>
<tr>
<td>endutxent</td>
<td>getservbyname</td>
<td>nl_langinfo</td>
<td>wctomb</td>
</tr>
<tr>
<td>getc_unlocked</td>
<td>getservbyport</td>
<td>ptsname</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 12.9** Functions *not* guaranteed to be thread-safe by POSIX.1
File Locks

- **ftrylockfile()** – Check lock associated with given FILE
  - `int ftrylockfile(FILE* fp);`

- **flockfile()** – Lock given file
  - `void flockfile(FILE* fp);`

- **funlockfile()** – Unlock given file
  - `void funlock(FILE* fp);`

- File lock is recursive, so locking while holding it does not cause a deadlock
Character-at-a-time Locking

- These functions should be surrounded by calls to flockfile() and funlockfile()

- `getchar_unlocked()` – get char from stdin
  - int `getchar_unlocked(void);`

- `getc_unlocked()` – get char from file `fp`
  - int `getc_unlocked(FILE* fp);`

- `putchar_unlocked()` – write char to stdout
  - int `putchar_unlocked(int c);`

- `putc_unlocked()` – write char to file `fp`
  - int `putc_unlocked(int c, FILE* fp);`
#include <limits.h>
#include <string.h>

#define MAXSTRINGSZ 4096

static char envbuf[MAXSTRINGSZ];
extern char **environ;

char *
getenv(const char *name)
{
    int i, len;

    len = strlen(name);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strcmp(name, environ[i], len) == 0) &&
            (environ[i][len] == '=')) {
            strcpy(envbuf, &environ[i][len+1], MAXSTRINGSZ-1);
            return(envbuf);
        }
    }
    return(NULL);
}

Figure 12.11  A nonreentrant version of getenv
Thread Specific Data

- We may want to keep data specific to a thread, which is hard in a shared address space
  - why?
    - errono and other process based interfaces
    - keeping private data

- We keep data private using a key:

- `pthread_key_create()` – Creates a key in order to store thread-private info
  - `int pthread-key_create(pthread_key_t* keyp, void (*destructor)(void *))`;

- Destructor is called when thread exits
Thread-Specific Data cont.

- **pthread_key_delete()** – Deletes key
  - `int pthread_key_delete(pthread_key_t key);`

- Note: Calling `pthread_key_delete()` does not invoke the destructor

- **pthread_once()** - solves race-conditions associated with keys
  - `pthread_once_t initflag = PTHREAD_ONCE_INIT;`
  - `int pthread_once(pthread_once_t* initflag, void (*initfn)(void));`
Cancel Options

- pthread_setcancelstate() – changes cancelability state of thread
  - int pthread_setcancelstate(int state, int* oldstate);
  - PTHREAD_CANCEL_ENABLE
  - PTHREAD_CANCEL_DISABLE

- pthread_testcancel() – add cancellation points to program
  - int pthread_testcancel(void);
  - if a cancellation is pending it will cancel thread
  - …unless cancellation is disabled
Cancel Options

- `pthread_setcanceltype()` – set the type of cancellation of the thread
  - `int pthread_setcancel(int type, int* oldtype);`
  - Types:
    - `PTHREAD_CANCEL_DEFERRED`
    - `PTHREAD_CANCEL_ASYNCHRONOUS`
Threads and Signals

- Each thread has its own signal mask
- Threads share signal disposition
  - All threads share signal handler actions
- `pthread_sigmask()` – examine and change blocked signal for a thread
  - `int pthread_sigmask(int how, const sigset_t* restrict set, sigset_t* restrict oset);`
Threads and Signals

- sigwait() – wait for a signal
  - int sigwait(const sigset_t* restrict set, int* restrict signop);

- pthread_kill() – kill for threads
  - int pthread_kill(pthread_t thread, int signo);
- **pthread_atfork()** – establish fork handlers to clean up and lock state
  - `pthread_atfork(void (*prepare),(void), void (*parent)(void),
  void (*child)(void));`
  - The prepare handler is called in the parent before the fork
    - Acquires all locks defined by parent
  - The parent handler is called in the parent after the fork before the child has returned
    - Unlocks all the clocks acquired by the prepare handler
  - The child handler is called before the child returns
    - Releases all locks acquired by the prepare handler