Threads

- Introduction
- POSIX Threads
- POSIX Threads on Linux
- Linux Memory Layout with Multiple Threads
- Example: Monitoring Multiple File Descriptors
- Example: Monitoring Multiple File Descriptors: Thread Function
- Example: Monitoring Multiple File Descriptors: `main()`
- Example: Monitoring Multiple File Descriptors: `main()` Continued
- Example: Monitoring Multiple File Descriptors: `main()` Continued
- Creating a Thread
- Thread Creation Continued
- Thread Creation Continued
- Waiting for a Thread
- Thread Identifiers
- Thread Termination Using `pthread_exit()`
- Bad Thread Program
- Bad Thread Program Continued
- Bad Thread Program Log

- Thread Termination
- Good Thread Program: `main()`
- Good Thread Program Log
- Clean-Up Handlers
- Passing Thread Result via Stack
- Thread Safety
- Thread Safety Examples
- Thread Safety Examples Continued
- Thread Attributes
- Attribute Initialization/Destruction
- Attribute Access
- Attribute Access Continued
- Detaching a Thread
- Thread Cancellation
- Cancelling a Thread
- Enabling / Disabling Cancellation State
- Set Cancellation Type
- Cancellation Points
- Thread Initialization
- Thread-Specific Data Keys
- Thread-Specific Data Keys Continued
- Accessing Thread Specific Data
- Thread Implementations
- Mutexes Creation/Destruction

- Initializing a Mutex Details
- Destroying a Mutex Details
- Locking a Mutex
- Mutex Unlocking
- Invariants, Critical Sections and Mutexes
- Mutex Lock Example
- Alarms Using Mutexes Global Defs
- Alarms Using Mutexes: Alarm Thread Routine
- Alarms Using Mutexes: Alarm Thread Routine Continued
- Alarms Using Mutexes: Main Routine
- Alarms Using Mutexes: Main Routine Continued
- Alarms Using Mutexes: Main Routine Continued
- Alarms Using Mutexes: Log
- Avoiding Deadlocks
- Reader-Writer Locks
- Reader-Writer Lock Initialization/Destruction
- Reader-Writer Locking
- Condition Variables
- Creating Condition Variables
- Waiting on a Condition Variable
- Using `pthread_cond_wait()`
- Waking Condition Variable Waiters
- Rules for Using Condition Variables

- Using Condition Variable Wait and Signal
- Invalid Wakeups
- Hello World
- Hello World Continued
- Hello World Revisited
- Hello World Revisited Continued
- Timed Wait on a Condition Variable
- Example Timed Wait
- Condition Variable Alarm: Global Defs
- Condition Variable Alarm: Alarm Insertion
- Condition Variable Alarm: Alarm Insertion Continued
- Condition Variable Alarm: Alarm Thread Routine
- Condition Variable Alarm: Alarm Thread Routine Continued
- Condition Variable Alarm: Alarm Thread Routine Continued
- Condition Variable Alarm: Main Routine
- Condition Variable Alarm: Main Routine Continued
- Condition Variable Alarm: Log
- Signals and Threads
- Signalling a Particular Thread
- Controlling Thread Signal Mask
- Signal Handling in MT Programs

- References

Introduction

- Threads execute **within** a process, sharing the **same** address space.

- Communication between threads is trivial, since they share the same address space.

- Synchronization between threads is necessary when they operate on shared data.

- If p is portion of program which can be speeded up, then Amdahl's law predicts a speedup of

$$\frac{1}{(1 - p) + p/n}$$

  with n processors.

- Useful even without multiple processors, when problem naturally solved using concurrent processes.

POSIX Threads

- Thread package integrated into POSIX.
- A large part of the POSIX interface is not thread-safe.
- Global `errno` is not thread-safe. POSIX supports a per thread `errno`.
- Almost all pthread functions return error number; `0` if ok.
- Need special versions of POSIX functions with non-thread-safe interface.

```
struct passwd *getpwnam(const char *name);

struct passwd *getpwnam_r(const char *name,
        struct passwd *pwd,  char *buffer, int buflen);
```

- Provides mutexes and condition variables for synchronization. Other synchronization primitives can be synthesized.

POSIX Threads on Linux

- Originally, Linux used a partial implementation of POSIX threads called *LinuxThreads*.
- Replaced by newer *Native POSIX Thread Library* NPTL.
- Uses `clone()` system-call and *futex*.

Linux Memory Layout with Multiple Threads

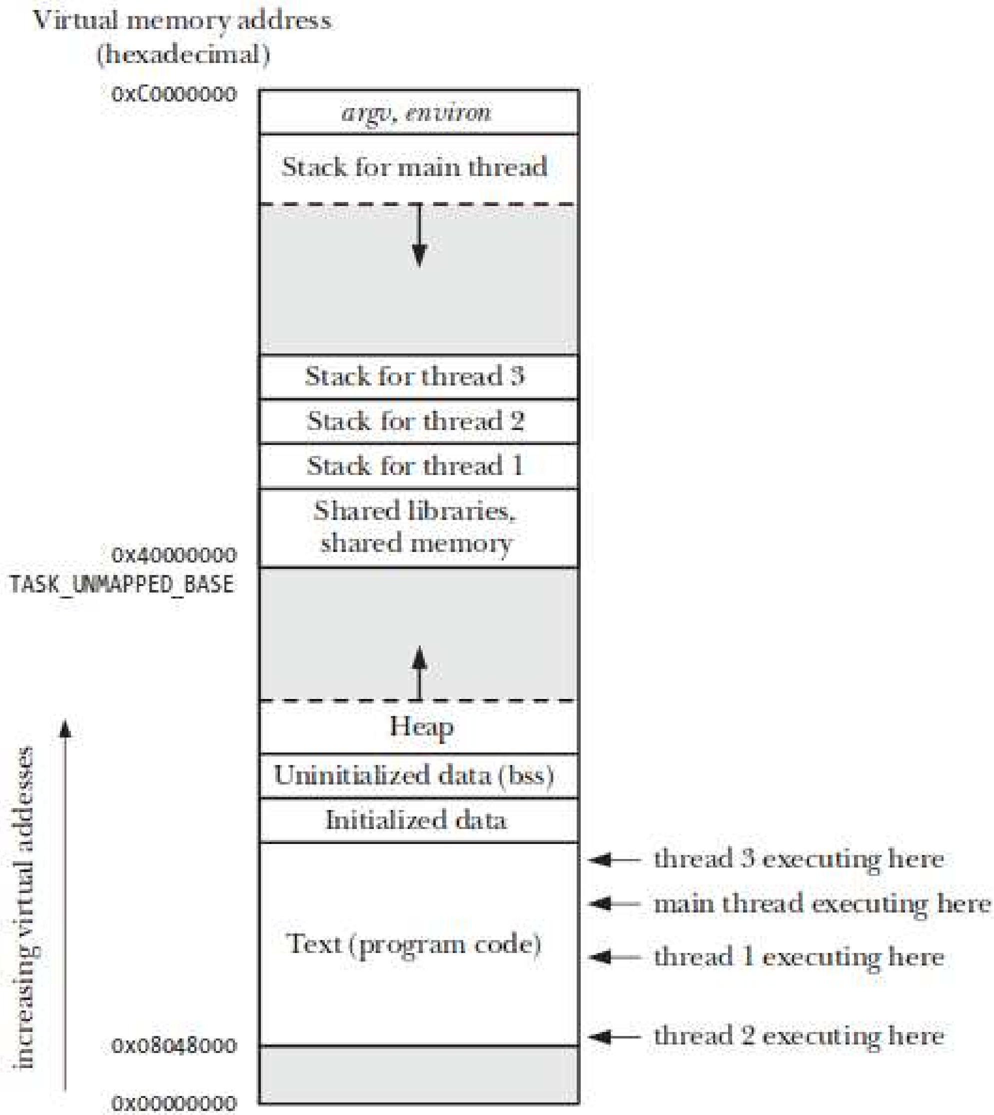

Figure 29-1: Four threads executing in a process (Linux/x86-32)

Example: Monitoring Multiple File Descriptors

Concurrently monitor multiple input file descriptors and respond to first descriptor which becomes ready. Possibilities:

- A separate process monitors each file descriptor.
- Use `select()`, or `poll()`.
- Use non-blocking I/O with polling.
- Use POSIX asynchronous I/O.
- Use separate threads for monitoring each descriptor.

Example: Monitoring Multiple File Descriptors: Thread Function

In ./programs/monitor-multiple-fds.c:

```
typedef struct {
  const char *fdName;
  int fd;
} ThreadArg;

/** Function called when thread started. */
static void *
processfd(void *arg)
{
  enum { MAX_BUF = 128 };
  char buf[MAX_BUF];
  const ThreadArg *tP = (const ThreadArg *)arg;
  for (;;) {
    ssize_t nbytes = read(tP->fd, buf, MAX_BUF);
    if (nbytes <= 0) break;
    printf("%s: %.*s", tP->fdName, (int)nbytes, buf);
  }
  return NULL;
}
```

Example: Monitoring Multiple File Descriptors: `main()`

```
int
main(int argc, char *argv[])
{
  if (argc != 2) {
    fprintf(stderr, "usage: %s FIFO\n", argv[0]);
    exit(1);
  }
  int fifoFD = open(argv[1], O_RDONLY);
  if (fifoFD < 0) {
    fprintf(stderr, "could not open %s: %s\n",
            argv[1], strerror(errno));
    exit(1);
  }
```

Example: Monitoring Multiple File Descriptors: `main()` Continued

```
enum { N_FDS = 2 };
ThreadArg args[N_FDS] = {
  { "STDIN", STDIN_FILENO },
  { "FIFO", fifoFD },
};
pthread_t tids[N_FDS];
for (int i = 0; i < N_FDS; i++) {
  int error =
    pthread_create(&tids[i], NULL, processfd, &args[i]);
  if (error != 0) {
    fprintf(stderr, "failed to create thread %d: %s\n",
            i, strerror(error));
    tids[i] = pthread_self();
  }
}
```

Example: Monitoring Multiple File Descriptors: `main()` Continued

```
  for (int i = 0; i < N_FDS; i++) {
    if (pthread_equal(pthread_self(), tids[i])) {
      continue;
    }
    else {
      int error = pthread_join(tids[i], NULL);
      if (error != 0) {
        fprintf(stderr, "failed to join thread %d: %s\n",
                i, strerror(error));
      }
    }
  }
  return 0;
}
```

Creating a Thread

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

`thread`

Pointer to `pthread_t` identifier.

`attr`

Attributes: detachment state, stack size, scheduling parameters.

`start`

Pointer to function at which thread starts. Thread terminates when function returns.

`arg`

Initial argument to function pointed to by `start`.

Thread Creation Continued

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

- Stores thread ID in `*thread`.
- Thread starts executing `start(arg)`. Ends when `start()` returns or calls `pthread_exit()`.

Thread Creation Continued

- If `attr` is `NULL`, creates thread with default attributes:
  - Unbound
  - Nondetached
  - With a default stack and stack size
  - With the parent’s priority
- Returns non-zero on error (`EAGAIN` for too many threads or `EINVAL` for bad `attr`).

Waiting for a Thread

```
int pthread_join(pthread_t tid, void **status);
```

- Blocks calling thread until the specified thread terminates.
- The specified thread must be in the current process and must not be detached.
- When status is not NULL, it points to a location that is set to the exit status of the terminated thread when `pthread_join`() returns successfully.
- Multiple threads cannot wait for the same thread to terminate. If they try to, one thread returns successfully and the others fail with an error of ESRCH.
- Returns non-zero on error (`ESRCH` `tid` not thread in current process; `EINVAL` bad `tid`; `EDEADLK` `tid` is calling thread!).

Thread Identifiers

```
pthread_t pthread_self(void);
int  pthread_equal(pthread_t tid1, pthread_t tid2);
```

- `pthread_self()` returns ID of calling thread.
- `pthread_equal()` returns non-zero if `tid1` and `tid2` are equal; zero otherwise. Results unpredictable if `tid1`, `tid2` not valid thread IDs.

Thread Termination Using `pthread_exit()`

```
void pthread_exit(void *status);
```

- Terminates the calling thread. All thread-specific data bindings are released. If the calling thread is not detached, then the thread's ID and the exit status specified by `status` are retained until the thread is waited for. Otherwise, `status` is ignored and the thread's ID can be reclaimed immediately.

- All cleanup handlers are executed in reverse order (the most recently pushed handler is run first). Finalization functions for non-NULL thread-specific values are called.

- Does not return. Instead the calling thread terminates with it's status set to `status` if non-NULL.

Bad Thread Program

Race condition in ./programs/bad-thread.c. Use `sleep()` to exhibit race:

```
static void *
threadFunction(void *arg)
{
  const char *msg = (const char *)arg;
  if (strlen(msg) > 5) sleep(1);
  write(STDOUT_FILENO, msg, strlen(msg));
  return NULL;
}
```

Bad Thread Program Continued

```
int
main(int argc, const char *argv[])
{
  if (argc != 3) {
    fprintf(stderr, "usage: %s MSG1 MSG2", argv[0]);
    exit(1);
  }
  const char *msg1 = argv[1];
  const char *msg2 = argv[2];
  write(STDOUT_FILENO, msg1, strlen(msg1));
  pthread_t tid;
  if (pthread_create(&tid, NULL, threadFunction, (void *)msg2) != 0) {
    fprintf(stderr, "cannot create thread: %s\n", strerror(errno));
    exit(1);
  }
  write(STDOUT_FILENO, "\n", 1);
  return 0;
}
```

Bad Thread Program Log

Different results based on whether or not `sleep()` is executed:

```
$ ./bad-thread hello world
hello
world$ ./bad-thread hello world1
hello
$
```

Thread Termination

- Thread terminates by returning from initial thread function.
- Thread terminates by explicitly calling `pthread_exit()`.
- Thread terminates by being *cancelled* by some thread in the same process.
- All threads in process terminate, if `exit()` is called explicitly or implicitly (by having `main()` return).
- If `main()` calls `pthread_exit()`, then only *main thread* is terminated. Process terminates only when **all** threads exit.

Good Thread Program: `main()`

Fix race condition by having main thread wait for auxiliary thread to terminate using `pthread_join()`. Same `threadFunction()` as before. In ./programs/good-thread.c:

```
int
main(int argc, const char *argv[])
{
  if (argc != 3) {
    fprintf(stderr, "usage: %s MSG1 MSG2", argv[0]);
    exit(1);
  }
  const char *msg1 = argv[1];
  const char *msg2 = argv[2];
  write(STDOUT_FILENO, msg1, strlen(msg1));
  pthread_t tid;
  if (pthread_create(&tid, NULL, threadFunction, (void *)msg2) != 0) {
    fprintf(stderr, "cannot create thread: %s\n", strerror(errno));
    exit(1);
  }
  if (pthread_join(tid, NULL) != 0) {
    fprintf(stderr, "cannot join thread: %s\n", strerror(errno));
    exit(1);
  }
  write(STDOUT_FILENO, "\n", 1);
  return 0;
}
```

Good Thread Program Log

```
$ ./good-thread hello world
helloworld
$ ./good-thread hello world1
helloworld1
$
```

Clean-Up Handlers

```
void pthread_cleanup_push(void(*routine)(void *), void *args);
void pthread_cleanup_pop(int execute);
```

- Allows registering of handlers to be run before thread-termination (run before thread-specific data destructors).
- Used for cleaning up allocated resources and restoring invariants.
- `pthread_cleanup_pop()` pops the last handler off the stack. It executes it iff `execute` is non-zero.

Passing Thread Result via Stack

```
void mainline (...)
{
  int result;
  pthread_attr_t tattr;
  pthread_t helper;
  int status;

  pthread_create(&helper, NULL, fetch, &result);

  /* do something else for a while */

  pthread_join(helper, &status);
  /* it’s now safe to use result */
}

void *fetch(void *resultv)
{
  int *result = (int *)resultv;
  /* fetch value from a database */

  *result = value;
  pthread_exit((void *)0);
}
```

Thread Safety

- Thread safety is an issue when order of thread access to data results in incorrect results.
- Three levels of thread safety:
    - Not thread safe.
    - Serializable.
    - MT-Safe.
- Asynch-Signal-Safe Functions: functions which can be called safely from a signal handler.

Thread Safety Examples

```
/* not thread-safe */
fputs(const char *s, FILE *stream) {
  char *p;
  for (p = s; *p; p++)
    putc((int)*p, stream);
}

/* serializable */
fputs(const char *s, FILE *stream) {
  static mutex_t m;
  char *p;
  mutex_lock(&m);
  for (p = s; *p; p++)
    putc((int)*p, stream);
  mutex_unlock(&m);
}
```

Thread Safety Examples Continued

```
/* MT-Safe */
mutex_t m[NFILE];
fputs(const char *s, FILE *stream) {
  char *p;
  mutex_lock(&m[fileno(stream)]);
  for (p = s; *p; p++)
    putc((int)*p, stream);
  mutex_unlock(&m[fileno(stream)]);
}
```

Thread Attributes

- Specified for thread on creation, or set later using attribute API.

- Attributes allow specifying:

  Scope

  Whether or not thread is bound to a LWP.

  Detach State

  Whether thread is detached.

  Stack Parameters

  Stack size and stack address.

  Priority

  Whether or not priority should be inherited from parent or set to some absolute priority.

  Scheduling Policy

  `SCHED_FIFO`, `SCHED_RR` (round-robin) or `SCHED_OTHER` (non-preemptive priority scheduling on Solaris).

Attribute Initialization/Destruction

```
int pthread_attr_init(pthread_attr_t *tattr);
int pthread_attr_destroy(pthread_attr_t *tattr);
```

- Default initialization values are: `PTHREAD_SCOPE_PROCESS` (unbound thread), `PTHREAD_CREATE_JOINABLE` (non-detached), 1 MB stack size at system-assigned address (NULL), priority inherited from parent, `SCHED_OTHER` scheduling.
- Returns non-zero on error. `pthead_attr_init()` returns ENOMEM if out of memory. `pthread_attr_destroy()` returns `EINVAL` if `tattr` is invalid.

Attribute Access

- `pthread_attr_setscope()`, `pthread_attr_getscope()`. Values `PTHREAD_SCOPE_PROCESS`, `PTHREAD_SCOPE_SYSTEM`.

- `pthread_attr_setdetachstate()`, `pthread_attr_getdetachstate()` Values `PTHREAD_CREATE_JOINABLE`, `PTHREAD_CREATE_DETACHED`.

- `pthread_attr_setschedpolicy()`, `pthread_attr_getschedpolicy()`. Values: `SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR` (latter 2 not supported for Realtime bound threads only in Solaris).

- `pthread_attr_setinheritsched()`, `pthread_attr_getinheritsched()`. Values: `PTHREAD_INHERIT_SCHED`, `PTHREAD_EXPLICIT_SCHED`.

Attribute Access Continued

- `pthread_attr_setschedparam()`, `pthread_attr_getschedparam()`. Only priority parameter supported.
- `pthread_attr_setstacksize()`, `pthread_attr_getstacksize()`.
- `pthread_attr_setstackaddr()`, `pthread_attr_getstackaddr()`.

Detaching a Thread

```
int pthread_detach(pthread_t tid);
```

- Used to specify that the storage for the thread `tid` can be reclaimed when the thread terminates.
- If `tid` has not terminated, `pthread_detach()` does not cause it to terminate.
- The effect of multiple `pthread_detach()` calls on the same target thread is unspecified.
- Returns non-zero on error (`ESRCH` `tid` thread not found; `EINVAL` thread not joinable).

Thread Cancellation

- A thread's cancel *state* (`PTHREAD_CANCEL_ENABLE`, `PTHREAD_CANCEL_DISABLE`) determines whether it can be cancelled by another thread.

- A thread's *cancel type* determines at which points in its execution a thread can be cancelled. `PTHREAD_CANCEL_ASYNCHRONOUS` means thread can be cancelled asynchronously at any point in its execution. `PTHREAD_CANCEL_DEFERED` means that the thread can be cancelled at only specific *cancellation points* in its execution.

- Cancelling threads in an improper state can lead to memory leakage and synchronization/deadlock errors.

- Use cleanup handlers to make sure that resources are released.

Cancelling a Thread

```
int pthread_cancel(pthread_t thread);
```

- By default, cancellation is deferred until `thread` reaches a cancellation point.
- Cancellation cleanup handlers for `thread` are called when the cancellation is acted on. Upon return of the last cancellation cleanup handler, the thread-specific data destructor functions are called for `thread`. `thread` is terminated when the last destructor function returns.
- Returns non-zero on error. ESRCH indicates invalid `thread`.

Enabling / Disabling Cancellation State

```
int pthread_setcancelstate(int state,
                           int *oldstate);
```

- `state` should be either `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`.
- `oldstate` allows nested cancellation states according to program logic.
- Returns non-zero on error. `EINVAL` if `state` is invalid.

Set Cancellation Type

```
int pthread_setcanceltype(int type,
                          int *oldtype);
```

- `type` should be either `PTHREAD_CANCEL_DEFER` (default) or `PTHREAD_CANCEL_ASYNCHRONOUS`.
- `oldtype` allows nested cancellation types according to program logic.
- Returns non-zero on error. `EINVAL` if `type` is invalid.

Cancellation Points

For *deferred cancellation*, cancellation points are:

- The programmatically-determined `pthread_testcancel()` call
- Threads waiting in `pthread_cond_wait()` or `pthread_cond_timedwait()`.
- Threads waiting for termination of another thread in `pthread_join()`.
- Threads blocked on `sigwait()`.
- Some standard library calls. In general, these are functions in which threads can block.

Thread Initialization

```
int  pthread_once(pthread_once_t *once_control,
                  void (*init_routine)(void));
```

- The purpose of `pthread_once()` is to ensure that a piece of initialization code is executed at most once. The `once_control` argument points to a static or extern variable statically initialized to `PTHREAD_ONCE_INIT`.

- The first time `pthread_once()` is called with `once_control` argument initialized to `PTHREAD_ONCE_INIT`, it calls `init_routine()` with no argument and changes the value of the `once_control` variable to record that initialization has been performed. Subsequent calls to `pthread_once()` with the same `once_control` argument do nothing.

- Returns non-zero on error. `EINVAL` indicates one or more `NULL` arguments.

- This call was necessary when mutexes could not be statically initialized. Now with static mutex initialization (`PTHREAD_MUTEX_INITIALIZER`) available, it’s functionality can be synthesized.

Thread-Specific Data Keys

```
int pthread_key_create(pthread_key_t *key,
                       void (*destructor) (void *));
int pthread_key_delete(pthread_key_t *key);
```

- `key` is global across all threads in process.
- `void *` value is local to each thread.
- If destructor function is non-NULL, then it is called for non-NULL values when thread terminates. Order of calling of destructor functions for multiple keys not specified.
- `pthread_key_delete()` does not call destructor functions.

Thread-Specific Data Keys Continued

- Limit of `PTHREAD_KEYS_MAX` keys available at any time.
- Returns non-zero on error. `pthread_key_create`() returns `ENOMEM` (out of memory) and `EAGAIN` (out of keys). `pthread_key_delete`() returns EINVAL for bad `key`.
- Similar concept in Java (since 1.2) using ThreadLocal.

Accessing Thread Specific Data

```
int pthread_setspecific(pthread_key_t key,
                        const void *value);
void *pthread_getspecific(pthread_key_t key);
```

- `pthread_setspecific()` returns non-zero on error (`ENOMEM` out of memory; `EINVAL` bad `key`).
- `pthread_getspecific()` returns thread-specific value associated with `key`; has no error return.
- If `pthread_setspecific()` replaces a previous heap-allocated value for a `key` with a new value, then a memory leak can occur.

Thread Implementations

- User-space thread libraries. Replace potentially blocking calls with *jacketed* versions.
- Kernel-space thread libraries.
- Hybrid thread models. Light-weight processes.
- Mixed-mode thread libraries.

Mutexes Creation/Destruction

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

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

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- `PTHREAD_MUTEX_INITIALIZER` used for initializing static mutexes.
- `pthread_mutex_init`() used for initializing dynamic mutexes.
- `pthread_mutex_destroy`() used for destroying mutexes after no threads are waiting on them.

Initializing a Mutex Details

```
int pthread_mutex_init(pthread_mutex_t *mp,
                       const pthread_mutexattr_t *mattr);
```

- If `mattr` is `NULL`, mutex initialized with `PTHREAD_PROCESS_PRIVATE`, meaning that mutex can only be used within the process (as opposed to `PTHREAD_PROCESS_SHARED` where mutex can be used in multi-processes).

- Alternatively if mutex is statically allocated, then it can be initialized by assigning `PTHREAD_MUTEX_INITIALIZER` to it (with default attributes).

- Non-zero return on error: `EBUSY`: The mutex cannot be reinitialized or modified because it still exists; `EINVAL`: The attribute value is invalid; `EAGAIN`: not enough resources to initialize another mutex; `ENOMEM`: not enough memory.

Destroying a Mutex Details

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

- Safe to destroy a mutex only if no threads are blocked on the mutex.
- No need to destroy a mutex initialized with `PTHREAD_MUTEX_INITIALIZER`.
- Space for storing the mutex is not freed by calling `pthread_mutex_destroy()`.
- Non-zero return on error:

  `EBUSY`
  : The specified mutex is locked or in use.

  `EINVAL`
  : The value specified by mp does not refer to an initialized mutex object.

Locking a Mutex

```
int pthread_mutex_lock(pthread_mutex_t *mp);
int pthread_mutex_trylock(pthread_mutex_t *mp);
```

- Only 1 thread can have a lock on a mutex at any time: hence guarantees MUTual EXclusion.
- `pthread_mutex_lock()` blocks till lock obtained, whereas `pthread_mutex_trylock()` returns with EBUSY if mutex is already locked.
- Returns non-zero on error:

  `EINVAL`

  `mp` does not refer to an initialized mutex object.

  `EBUSY`

  `mp` already locked (for `pthread_mutex_trylock()`).

Mutex Unlocking

```
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

- The calling thread **must** hold a lock on the mutex pointed to by `mp`.
- If other threads are waiting for the mutex, the thread at the head of the queue is unblocked.
- Returns non-zero on error:

  `EINVAL`
  : The value specified by `mp` does not refer to an initialized mutex object.

Invariants, Critical Sections and Mutexes

- *Invariants* are assumptions about the *state* of a program. Example: a queue header is null (the queue is empty) or points to the first element of the queue.

- *Critical sections* are sections of code which temporarily break an invariant. Example: inserting an element onto the head of a queue.

- *Mutexes* guarantee that mutual exclusion of multiple threads from critical sections, guaranteeing that invariants are preserved.

Mutex Lock Example

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;

void
increment_count()
{
  pthread_mutex_lock(&count_mutex);
  count = count + 1;
  pthread_mutex_unlock(&count_mutex);
}

long long
get_count()
{
  long long c;

  pthread_mutex_lock(&count_mutex);
  c = count;
  pthread_mutex_unlock(&count_mutex);
  return (c);
}
```

Alarms Using Mutexes Global Defs

In ./programs/mutex-alarm.c:

```
enum { MESSAGE_SIZE = 64 };

typedef struct Alarm {
  struct Alarm *succ;
  int seconds;
  time_t time;
  char message[MESSAGE_SIZE];
} Alarm;

static Alarm *alarmList = NULL;
static pthread_mutex_t alarmListMutex=
  PTHREAD_MUTEX_INITIALIZER;
```

Alarms Using Mutexes: Alarm Thread Routine

```
/* Alarm thread start routine */
static void *
alarmThread(void *arg)
{
  while (1) {
    int sleepTime;
    Alarm *alarmP;
    int status = pthread_mutex_lock(&alarmListMutex);
    if (status != 0) {
      perror("mutex lock"); exit(1);
    }
    alarmP = alarmList;
    if (alarmP == NULL) {
      sleepTime = 1;
    }
    else {
      time_t now = time(NULL);
      alarmList = alarmP->succ;
      sleepTime = (alarmP->time <= now)
                 ? 0
                 : (alarmP->time - now);
    }
    status = pthread_mutex_unlock(&alarmListMutex);
```

Alarms Using Mutexes: Alarm Thread Routine Continued

```
    if (status != 0) {
      perror("mutex unlock"); exit(1);
    }
    if (sleepTime > 0) {
      sleep(sleepTime);
    }
    else {
      sched_yield();
    }
    if (alarmP != NULL) {
      printf("(%d) %s\n", alarmP->seconds,
             alarmP->message);
      free(alarmP);
    }
  }
}
```

Alarms Using Mutexes: Main Routine

```
int
main(int argc, char *argv[])
{
  pthread_t alarmThreadID;
  int status = pthread_create(&alarmThreadID, NULL,
                              alarmThread, NULL);
  if (status != 0) {
    perror("thread create"); exit(1);
  }

  while (1) {
    enum { LINE_SIZE = 128 };
    char line[LINE_SIZE];
    printf("alarm> "); fflush(stdout);
    if (!fgets(line, LINE_SIZE, stdin)) exit(0);
    if (strlen(line) <= 1) continue;
    if (line[strlen(line) - 1] != '\n') {
      fprintf(stderr, "input too long\n");
      continue;
    }
```

Alarms Using Mutexes: Main Routine Continued

```
else {
  Alarm *alarmP = malloc(sizeof(Alarm));
  if (!alarmP) {
    perror("malloc"); exit(1);
  }
  if (sscanf(line, "%d %64[^\n]",
             &alarmP->seconds,
             alarmP->message) != 2) {
    fprintf(stderr, "bad input\n");
    free(alarmP);
    continue;
  }
  alarmP->time =
    time(NULL) + alarmP->seconds;
  if (pthread_mutex_lock(&alarmListMutex)
      != 0) {
    perror("main mutex lock"); exit(1);
  }
```

Alarms Using Mutexes: Main Routine Continued

```
      { Alarm **last = &alarmList;
        Alarm *succ = *last;
        while (succ != NULL) {
          if (succ->time >= alarmP->time) {
            alarmP->succ = succ;
            *last = alarmP;
            break;
          }
          last = &succ->succ;
          succ = succ->succ;
        }
        if (succ == NULL) {
          *last = alarmP; alarmP->succ = NULL;
        }
      }
      if (pthread_mutex_unlock(&alarmListMutex)
          != 0) {
        perror("main mutex unlock"); exit(1);
      }
    }
  }
}
```

Alarms Using Mutexes: Log

```
$ ./mutex-alarm
alarm> 10 msg1
alarm> 20 msg2
alarm> (10) msg1
(20) msg2

alarm> 30 msg3
alarm> 10 msg4
alarm> (30) msg3
(10) msg4

alarm> $
```

Avoiding Deadlocks

- If a thread locks mutex A what happens if it attempts a further lock on A? If mutex is *recursive*, then it keeps a lock count and everything works ok. Otherwise, depending on the implementation, a **deadlock** may occur.
- If thread 1 acquires mutex A followed by mutex B, whereas thread 2 acquires mutex B followed by mutex A, then a deadlock can occur.
- One way of avoiding deadlocks is to define a total ordering among mutexes and to always have all threads acquire mutexes in that order.
- If total ordering of mutexes is not possible, have each thread acquire its first lock unconditionally (using `pthread_mutex_lock()`) and acquire subsequent locks conditionally (using `pthread_mutex_trylock()`). If the try-lock fails, then the thread should release all locks and attempt to reacquire locks after some random time delay.

Reader-Writer Locks

- Reader-writer locks allow higher-levels of parallelism than mutexes.
- Mutex allows only two states: locked or unlocked, with only 1 thread holding the lock.
- Reader-writer locks have three states: read-locked, write-locked, unlocked. Only one thread can be in write-locked, but multiple threads can be in read-lock.
- Also claeed shared-exclusive locks, with shared read-lock, but exclusive write-lock.

Reader-Writer Lock Initialization/Destruction

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

- Can pass `NULL` pointer for `attr` for default attributes (only standard supported attribute is *process-shared*).
- Must call `pthread_rwlock_destroy()` before freeing underlying memory occupied by lock to avoid resource-leak.

Reader-Writer Locking

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

- Return 0 if ok, `errno` on failure (test versions return EBUSY if lock can't be acquired because another thread has the lock).
- There may be a implementation-defined limit on the maximum number of simultaneous read locks.

Condition Variables

- Used for communicating information about the state of shared data.
- Condition variables are used for *signalling*, not for mutual exclusion.
- Used in conjunction with a mutex.
- Mutex must be locked before condition variable wait.
- Mutex is released during wait and then reacquired when wait returns.
- A condition variable wait always returns with the mutex locked.
- A mutex may be associated with more than 1 condition variable (a queue may be *full* or *empty*).
- A condition variable should be associated with only 1 mutex.

Creating Condition Variables

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);
```

- `PTHREAD_COND_INITIALIZER` used for initializing static condition variables.
- `pthread_cond_init`() used for initializing dynamic condition variables.
- `pthread_cond_destroy`() used for destroying condition variables after no threads are waiting on them.
- Non-zero return on error: `EBUSY`: The condition variable is being used; `EINVAL`: The attribute value is invalid; `EAGAIN`: not enough resources to initialize another condition variable; `ENOMEM`: not enough memory.

Waiting on a Condition Variable

```
int pthread_cond_wait(pthread_cond_t *cv,
                      pthread_mutex_t *mutex);
```

- The blocked thread can be awakened by a `pthread_cond_signal()`, a `pthread_cond_broadcast()`, or when interrupted by delivery of a signal.
- Any change in the value of a condition associated with the condition variable cannot be inferred by the return of `pthread_cond_wait()`, and any such condition must be reevaluated.
- The `pthread_cond_wait()` routine always returns with the mutex locked and owned by the calling thread even when returning an error.
- Returns non-zero on error: `EINVAL` for bad argument.

Using `pthread_cond_wait()`

- blocks until the condition is signaled; atomically releases the associated mutex lock before blocking, and atomically reacquires it before returning.

- Typically, a condition expression is evaluated under the protection of a mutex lock. When the condition expression is false, the thread blocks on the condition variable. The condition variable is then signaled by another thread when it changes the condition value. This causes one or all of the threads waiting on the condition to unblock and to try to reacquire the mutex lock.

- Because the condition can change before an awakened thread returns from `pthread_cond_wait()`, the condition that caused the wait must be retested before the mutex lock is acquired: enclose `pthread_cond_wait()` in a `while`-loop.

Waking Condition Variable Waiters

```
int pthread_cond_signal(pthread_cond_t *cv);
int pthread_cond_broadcast(pthread_cond_t *cv);
```

- Signal under the protection of the same mutex used with the condition variable being signaled. Otherwise, the condition variable could be signaled between the test of the associated condition and blocking in `pthread_cond_wait()`, which can cause an infinite wait.

- Scheduling policy determines wake-up order of blocked threads. For `SCHED_OTHER`, threads are awakened in priority order.

- If no threads are blocked on `cv`, then no effect.

- Returns non-zero on error: `EINVAL` bad `cv`.

Rules for Using Condition Variables

- Acquire mutex before testing predicate.
- Retest predicate after returning from `pthread_cond_wait()` because predicate need not necessarily have become true or it could be a false wakeup.
- Acquire the mutex before changing any of the variables appearing in the predicate, or using `pthread_cond_signal()` or `pthread_cond_broadcast()`.
- Hold the mutex only for a short period of time - usually while testing the predicate. Release the mutex ASAP either explicitly with `pthread_mutex_unlock()` or implicitly with `pthread_cond_wait()`.

Using Condition Variable Wait and Signal

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned int 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);
  count = count + 1;
  if (count == 1) {
    pthread_cond_signal(&count_nonzero);
  }
  pthread_mutex_unlock(&count_lock);
}
```

Invalid Wakeups

Invalid wakeup condition can be caused by:

Intercepted wakeups

Some other thread locks the mutex first and makes the condition false before this thread gets the mutex.

Loose predicates

Programatically, it may be easier to signal the condition variable when a weaker condition is true.

Spurious wakeups

On some architectures, it is difficult to guarantee that spurious signalling will not occur. Hence it is the responsibility of the programmer to ensure that the condition is indeed true.

Hello World

In ./programs/hello-world.c:

```
static char *messages[] = { "Hello", "World" };

int main()
{
  pthread_t thread1, thread2;

  pthread_create(&thread1, NULL,
                 &print_message_function,
                 (void *)0);
  pthread_create(&thread2, NULL,
                 &print_message_function,
                 (void *)1);

  return 0;
}
```

Hello World Continued

```
void *print_message_function(void *ptr)
{
  int messageN = (intptr_t)ptr;
  char *message = messages[messageN];
  printf("%s ", message);
  return NULL;
}
```

Hello World Revisited

```
static char *messages[] = { "Hello", "World" };

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static int state = 0;

static void waitTillState(int s) {
  pthread_mutex_lock(&mutex);
  while (state < s) {
    pthread_cond_wait(&cond, &mutex);
  }
  pthread_mutex_unlock(&mutex);
}

static void nextState(int s) {
  pthread_mutex_lock(&mutex);
  state = s;
  pthread_cond_broadcast(&cond);
  pthread_mutex_unlock(&mutex);
}
```

Hello World Revisited Continued

```
int main()
{
  pthread_t thread1, thread2;

  pthread_create(&thread1, NULL,
                 print_message_function,
                 (void *)0);
  pthread_create(&thread2, NULL,
                 print_message_function,
                 (void *)1);
  waitTillState(2);
  return 0;
}
static void *print_message_function(void *ptr)
{
  int messageN = (intptr_t)ptr;
  char *message = messages[messageN];
  waitTillState(messageN);
  printf("%s ", message);
  nextState(messageN + 1);
  return NULL;
}
```

Timed Wait on a Condition Variable

```
int pthread_cond_timedwait(pthread_cond_t *cv,
                           pthread_mutex_t *mp,
                           const struct timespec *abstime);
```

- Does not block past absolute time specified by `abstime`.
- The time-out is specified as a time of day so that the condition can be retested efficiently without recomputing the value.
- Returns non-zero on error:

  `EINVAL`
  : `cv` or `abstime` points to an illegal address.

  `ETIMEDOUT`
  : The time specified by `abstime` has passed.

Example Timed Wait

```
pthread_timestruc_t to;
pthread_mutex_t m;
pthread_cond_t c;
...
pthread_mutex_lock(&m);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;
while (cond == FALSE) {
  err = pthread_cond_timedwait(&c, &m, &to);
  if (err == ETIMEDOUT) {
    /* timeout, do something */
    break;
  }
}
pthread_mutex_unlock(&m);
```

Condition Variable Alarm: Global Defs

```
enum { MESSAGE_SIZE = 64 };

typedef struct Alarm {
  struct Alarm *succ;
  int seconds;
  time_t time;
  char message[MESSAGE_SIZE];
} Alarm;

static Alarm *alarmList = NULL;
time_t currentAlarm = 0;
static pthread_mutex_t alarmListMutex=
  PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t alarmListCond=
  PTHREAD_COND_INITIALIZER;
```

Condition Variable Alarm: Alarm Insertion

```
/* Called with alarmListMutex locked */
static void
alarmInsert(Alarm *alarmP)
{
  Alarm **last = &alarmList;
  Alarm *succ = *last;
  while (succ != NULL) {
    if (succ->time >= alarmP->time) {
      alarmP->succ = succ;
      *last = alarmP;
      break;
    }
    last = &succ->succ;
    succ = succ->succ;
  }
  if (succ == NULL) {
    *last = alarmP; alarmP->succ = NULL;
  }
```

Condition Variable Alarm: Alarm Insertion Continued

```
  if (currentAlarm == 0 ||
      alarmP->time < currentAlarm) {
    currentAlarm = alarmP->time;
    if (pthread_cond_signal(&alarmListCond)) {
      perror("signal"); exit(1);
    }
  }
}
```

Condition Variable Alarm: Alarm Thread Routine

```
/* Alarm thread start routine */
static void *
alarmThread(void *arg)
{
  int status = pthread_mutex_lock(&alarmListMutex);
  if (status != 0) {
    perror("mutex lock"); exit(1);
  }
  while (1) {
    Alarm *alarmP;
    int expired;
    time_t now;
    currentAlarm = 0;
    while (alarmList == NULL) {
      if (pthread_cond_wait(&alarmListCond,
                            &alarmListMutex)) {
        perror("cond wait"); exit(1);
      }
    }
    alarmP = alarmList;
    alarmList = alarmP->succ;
    now = time(NULL);
    expired = 0;
```

Condition Variable Alarm: Alarm Thread Routine Continued

```
if (alarmP->time > now) {
  struct timespec condTime;
  condTime.tv_sec = alarmP->time;
  condTime.tv_nsec = 0;
  currentAlarm = alarmP->time;
  while (currentAlarm == alarmP->time) {
    int status =
      pthread_cond_timedwait(&alarmListCond,
                             &alarmListMutex,
                             &condTime);
    if (status == ETIMEDOUT) {
      expired = 1; break;
    }
    if (status != 0) {
      perror("timed wait"); exit(1);
    }
  }
  if (!expired) alarmInsert(alarmP);
}
```

Condition Variable Alarm: Alarm Thread Routine Continued

```
      else {
        expired = 1;
      }
      if (expired) {
        printf("(%d) %s\n", alarmP->seconds,
                            alarmP->message);
        free(alarmP);
      }
    }
}
```

Condition Variable Alarm: Main Routine

```
int
main(int argc, char *argv[])
{
  pthread_t alarmThreadID;
  int status = pthread_create(&alarmThreadID,
                              NULL,
                              alarmThread, NULL);
  if (status != 0) {
    perror("thread create"); exit(1);
  }

  while (1) {
    enum { LINE_SIZE = 128 };
    char line[LINE_SIZE];
    printf("alarm> "); fflush(stdout);
    if (!fgets(line, LINE_SIZE, stdin)) exit(0);
    if (strlen(line) <= 1) continue;
    if (line[strlen(line) - 1] != '\n') {
      fprintf(stderr, "input too long\n");
      continue;
    }
```

Condition Variable Alarm: Main Routine Continued

```
    else {
      Alarm *alarmP = malloc(sizeof(Alarm));
      if (!alarmP) { perror("malloc"); exit(1); }
      if (sscanf(line, "%d %64[^\n]",
                 &alarmP->seconds,
                 alarmP->message) != 2) {
        fprintf(stderr, "bad input\n");
        free(alarmP);
        continue;
      }
      alarmP->time =
        time(NULL) + alarmP->seconds;
      if (pthread_mutex_lock(&alarmListMutex) != 0) {
        perror("main mutex lock"); exit(1);
      }
      alarmInsert(alarmP);
      if (pthread_mutex_unlock(&alarmListMutex) != 0) {
        perror("main mutex unlock"); exit(1);
      }
    }
  }
}
```

Condition Variable Alarm: Log

```
$ ./cond-alarm
alarm> 20 msg1
alarm> 10 msg2
alarm> 5 msg3
alarm> (5) msg3
(10) msg2
(20) msg1

alarm> $
```

Signals and Threads

- Signal handlers are process wide.
- Each thread has its own signal mask.
- Asynchronous signals are delivered to any thread which has it unmasked.
- Synchronous signals are delivered to causing thread,
- It is possible to direct a signal to a particular thread using `pthread_kill`().

Signalling a Particular Thread

```
int pthread_kill(pthread_t thread, int sig);
```

- Send signal number `sig` to specified `thread`.
- Success returns 0. `EINVAL` for invalid `sig`; `ESRCH` for bad `thread`.

Note that

```
pthread_kill(pthread_self(), SIGKILL)
```

is guaranteed to kill the **entire** process.

Controlling Thread Signal Mask

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
```

- `how` of `SIG_SETMASK` sets mask to `set`; `SIG_BLOCK` adds `set` to current mask; `SIG_UNBLOCK` removes signals in `set` from current mask.
- If `oset` is not NULL, returns previous mask in `oset`.
- Return 0 on success; `EINVAL` if bad `how`.

Signal Handling in MT Programs

- Have main program mask off all signals.
- Designate dedicated threads for signal handling.
- Have signal handler thread unmask handled signals or using `sigwait()`.

References

- Text, Ch. 29 - 33.
- APUE, Ch 11 and 12.
- Solaris 2.5 Software Developer AnswerBook: *Multithreaded Programming Guide*. Available online. Watch out for errors.
- FSF, *The GNU C Library*, at http://www.gnu.org/software/libc/manual/.
- John Ousterhout, *Why Threads Are A Bad Idea*, at http://www.csd.uoc.gr/~hy527/papers/threads-ousterhout.pdf.
- David R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.
- Bill Lewis, Daniel J. Berg, *Threads Primer: A Guide to Multithreaded Programming*, Prentice-Hall, 1996.

- Bradford Nichols, Dick Buttlar & Jacqueline Proulx Farrell, *Pthreads Programming*, O'Reilly, 1996.