

# Pthreads

CS 498: Compiler Optimizations  
Fall 2006

University of Illinois at Urbana-Champaign



## Threads vs. processes (created with fork)

Property	Processes created with fork	Threads of a process	Ordinary function calls
variables	get copies of all variables	share global variables	share global variables
IDs	get new process IDs	share the same process ID but have unique thread ID	share the same process ID (and thread ID)
Communication	Must explicitly communicate, e.g. pipes or use small integer return value	May communicate with return value or shared variables if done carefully	May communicate with return value or shared variables (don't have to be careful)
Parallelism (one CPU)	Concurrent	Concurrent	Sequential
Parallelism (multiple CPUs)	May be executed simultaneously	Kernel threads may be executed simultaneously	Sequential



# Thread components

---

A thread has its own program counter and stack, but shares a number of resources with its process and other threads of the process:

- address space: code and global variables
- open files
- signals
- timers
- process ID



## Pthreads--- POSIX Threads

---

- Pthreads: P from POSIX (Portable Operating System Interface)
- It is a standard API
  - Supported by most vendors
  - General concepts applicable to other thread APIs (java threads, NT threads,etc).
- Low level functions
  - No high level constructs



# What's POSIX Got To Do With It?

---

- Each OS had it's own thread library and style
- That made writing multithreaded programs difficult because:
  - you had to learn a new API with each new OS
  - you had to *modify your code* with each port to a new OS
- POSIX (IEEE 1003.1c-1995) provided a standard known as Pthreads
- Unix International (UI) threads (Solaris threads) are available on Solaris (which also supports POSIX threads)



## Pthread Operations

---

POSIX function	description
<code>pthread_cancel</code>	terminate another thread
<code>pthread_create</code>	create a thread
<code>pthread_detach</code>	set thread to release resources
<code>pthread_equal</code>	test two thread IDs for equality
<code>pthread_exit</code>	exit a thread without exiting process
<code>pthread_kill</code>	send a signal to a thread
<code>pthread_join</code>	wait for a thread
<code>pthread_self</code>	find out own thread ID



# Return Values

---

- Most POSIX functions return 0 on success and a nonzero error code on failure.
  - They do not set errno but the value returned when an error occurs has the value that errno would have.



# Creating a Thread

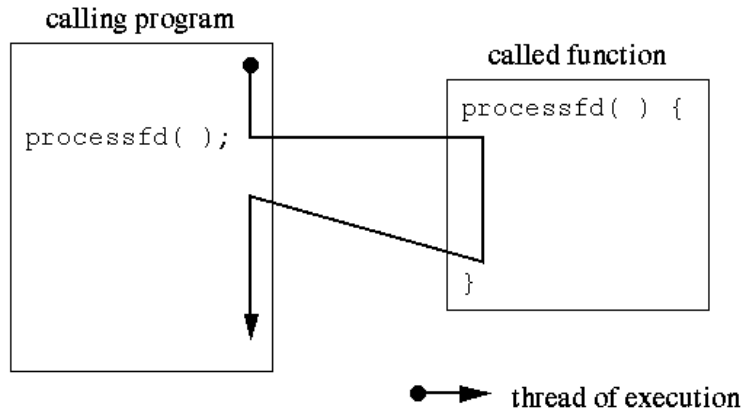
---

- When a new thread is created it runs concurrently with the creating process.
- When creating a thread you indicate which function the thread should execute.



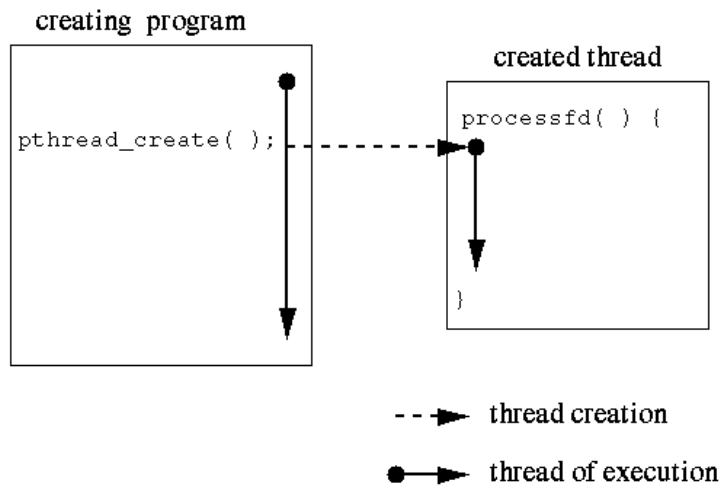
# Normal function call

---



# Threaded function call

---



# Creating a thread

---

A thread is created with `pthread_create`

```
int pthread_create(  
    pthread_t *thread_id,  
    const pthread_attr_t *attr,  
    void *(*start_routine)(void *),  
    void *args );
```

- Thread ID - Calling thread must provide location for this
- Thread attributes - `NULL` pointer indicates default attributes
- Function for thread to run
- Arguments



## Example thread creation

---

```
main(void)  
{  
    pthread_t thread1, thread2;  
  
    pthread_create(&thread1,  
        NULL,  
        (void *) do_one_thing,  
        (void *) &r1);  
  
    pthread_create(&thread2,  
        NULL,  
        (void *) do_another_thing,  
        (void *) &r2);  
    pthread_join(thread1, NULL);  
    pthread_join(thread2, NULL);  
  
    return 0;  
}
```



# Creating a thread

---

A thread is created with

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



## Restrict Keyword

---

- One of the new features in the recently approved C standard C99
- This qualifier can be applied to a data pointer to indicate that
  - During the scope of that pointer declaration, all data accessed through **it will be accessed only through that pointer but not through any other pointer.**
  - It enables the compiler to perform certain optimizations based on the premise that a given object cannot be changed through another pointer
    - Can avoid pointer aliasing problem



# Threaded Function Call Detail

---

- A function that is used as a thread must have a special format.
  - It takes a single parameter of type `pointer to void` and returns a `pointer to void`.
    - The parameter type allows any pointer to be passed.
    - This can point to a structure, so in effect, the function can use any number of parameters.



## The Thread ID

---

`pthread_t pthread_self(void)`

- Each thread has an id of type `pthread_t`.
  - On most systems this is just an integer (like a process ID)
  - But it does not have to be
- A thread can get its ID with `pthread_self`
- Compare two threads
  - `int pthread_equal(pthread_t t1, pthread_t t2)`



# Joining Threads

---

- Suspends caller until specified thread exits
  - Similar to `waitpid` for processes
- Good practice – call `pthread_detach` or `pthread_join` for every thread
- `pthread_join` gets value passed to `pthread_exit` by terminating thread



# Joining Threads

---

```
void *copyfilemalloc(void *arg);
char *bytesptr = (char *) malloc(MAX_BYTES);
pthread_t tid = -1;

if (((fds[0] = open(argv[1], READ_FLAGS)) == -1) ||
    ((fds[1] = open(argv[2], WRITE_FLAGS, PERMS)) == -1)) {
    perror("Failed to open the files");
    return 1;
}
if (error = pthread_create(&tid, NULL, copyfilemalloc, (void *)fds)) {
    fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
    return 1;
}
if (error = pthread_join(tid, (void **)&bytesptr)) {
    fprintf(stderr, "Failed to join thread: %s\n", strerror(error));
    return 1;
}
```



# Thread Detach & Join

---

- Call `pthread_join()` or `pthread_detach()` for every thread that is created joinable
  - so that the system can reclaim all resources associated with the thread
- Failure to join or to detach threads → memory and other resource leaks until the process ends



## Detaching a Thread

---

`int pthread_detach(pthread_t threadid);`

- Indicate that system resources for the specified thread should be reclaimed when the thread ends
  - If the thread is already ended, resources are reclaimed immediately
  - This routine does not cause the thread to end
- A detached thread's thread ID is undetermined
- Threads are detached
  - after a `pthread_detach()` call
  - after a `pthread_join()` call
  - if a thread terminates and the `PTHREAD_CREATE_DETACHED` attribute was set on creation



# How to make a thread detached

---

```
void *processfd(void *arg);

int error;
int fd
pthread_t tid;

if (error = pthread_create(&tid, NULL, processfd, &fd)) {
    fprintf(stderr, "Failed to create thread: %s\n",
strerror(error));
}
else if (error = pthread_detach(tid)){
    fprintf(stderr, "Failed to detach thread: %s\n",
strerror(error));
}
```



# How a thread can detach itself

---

```
void *detachfun(void *arg) {
    int i = *((int *)(arg));

    if (!pthread_detach(pthread_self()))
        return NULL;

    fprintf(stderr, "My argument is %d\n", i);
    return NULL;
}
```



## Another example: Detaching Threads

---

- Detach a thread to make it release its resources upon exiting
- Detached thread cannot be joined.

```
void *processfd(void *arg);

int error;
int fd
pthread_t tid;

if (error = pthread_create(&tid, NULL, processfd, &fd))
    fprintf(stderr, "Failed to create thread: %s\n",
    strerror(error));
else if (error = pthread_detach(tid))
    fprintf(stderr, "Failed to detach thread: %s\n",
    strerror(error));
```



- Make a thread detach itself
  - Replace `tid` with `pthread_self()`

## “Waiting” on a Thread: `pthread_join()`

---

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

- `pthread_join()` is a blocking call on **non-detached** threads
- It indicates that the caller wishes to block until the thread being joined exits
- You cannot join on a detached thread, only non-detached threads (detaching means you are NOT interested in knowing about the threads exit)



# Pthread\_join

---

```
int error;
int *exitcodep;
pthread_t tid;

if (error = pthread_join(tid, &exitcodep)){
    fprintf(stderr, "Failed to join thread: %s\n", strerror(error));
}
else {
    fprintf(stderr, "The exit code was %d\n", *exitcodep);
}
```



## Exiting and Cancellation

---

- Question:
  - If a thread calls exit(), what about other threads in the same process?
- A process can terminate when:
  - it calls exit directly
  - one of its threads calls exit
  - it executes return from main
  - it receives a termination signal
- In any of these cases, all threads of the process terminate.



# Exiting

---

When a thread is done, it can call return from its main function (the one used by `pthread_create`) or it can call `pthread_exit`

```
void pthread_exit(void *value_ptr);
```

One thread can request that another exit with `pthread_cancel`

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

The `pthread_cancel` returns after making the request.

A successful return does not mean that the target thread has terminated or even that it eventually will terminate as a result of the request



## Thread Exit: Example

---

```
#include <pthread.h>
int theStatus=5;
void *threadfunc(void *parm){
    printf("Inside secondary thread\n");
    pthread_exit(__VOID(theStatus));
}
int main(int argc, char **argv){
    pthread_t thread;
    int rc=0; void *status;
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    rc = pthread_join(thread, &status);
    checkResults("pthread_join()\n", rc);
    if (__INT(status) != theStatus) {
        printf("Secondary thread failed\n"); exit(1);}
    printf("Got secondary thread status as expected\n");
    printf("Main completed\n");
    return 0;}
```



# Thread Cancellation

---

One thread can request that another exit with `pthread_cancel`

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

The `pthread_cancel` returns after making the request.



# Thread Cancellation

---

- The cancellation state of a thread determines the action taken upon receipt of a cancel request.
  - The cancellation state can have two values:
    - `PTHREAD_CANCEL_ENABLE`
    - `PTHREAD_CANCEL_DISABLE`
    - Can be set using `pthread_setcancelstate()`
  - When the state is `PTHREAD_CANCEL_ENABLE`, the cancellation type can be:
    - `PTHREAD_CANCEL_ASYNCHRONOUS`
    - `PTHREAD_CANCEL_DEFERRED`
    - Can be set using `pthread_setcanceltype()`
- If a thread receives a cancel, when the cancel state is `DISABLE`, or the type is `DEFERRED`, the cancel is held pending in the target thread until the state changes to `ENABLE`, and the type to `ASYNCHRONOUS`.



# Exiting and Cancelling

---

- Process & all threads terminate when:
  - The process or any thread calls `exit`
  - The process executes `return` from `main`
  - It receives a signal
- To exit a single thread:

```
void pthread_exit(void *value_ptr);
```
- Main thread should block (`join`) or call `pthread_exit(NULL)` to wait for all threads
- Threads can request another thread to be cancelled by:

```
void pthread_cancel(pthread_t thread);
```



## Table 12.1

---

POSIX function	Description
<code>pthread_cancel</code>	terminate another thread
<code>pthread_create</code>	create a thread
<code>pthread_detach</code>	set thread to release resources
<code>pthread_equal</code>	test two thread IDs for equality
<code>pthread_exit</code>	exit a thread without exiting process
<code>pthread_kill</code>	send a signal to a thread
<code>pthread_join</code>	wait for a thread
<code>pthread_self</code>	find out own thread ID



## Passing parameters and returning values

---

You can pass multiple parameters to a thread by passing a pointer, such as an array, to the thread when it is created.

Care must be taken in receiving return values:

- The terminating thread passes a pointer to the joining thread
- They share the same address space.
- The return value must exist after the thread terminates
  - You cannot use an automatic variable in the thread for the return value



## Thread Attributes

---

- Create an attribute object (initialize it with default properties)
- Modify the properties of the attribute object
- Create a thread using the attribute object
- The object can be changed or reused without affecting the thread
- The attribute object affects the thread only at the time of thread creation



# Settable properties of thread attributes

---

property	function
attribute objects	<code>pthread_attr_destroy</code>
	<code>pthread_attr_init</code>
state	<code>pthread_attr_getdetachstate</code>
	<code>pthread_attr_setdetachstate</code>
stack	<code>pthread_attr_getguardsize</code>
	<code>pthread_attr_setguardsize</code>
	<code>pthread_attr_getstack</code>
	<code>pthread_attr_setstack</code>
scheduling	<code>pthread_attr_getinheritsched</code>
	<code>pthread_attr_setinheritsched</code>
	<code>pthread_attr_getschedparam</code>
	<code>pthread_attr_setschedparam</code>
	<code>pthread_attr_getschedpolicy</code>
	<code>pthread_attr_setschedpolicy</code>
	<code>pthread_attr_getscope</code>
	<code>pthread_attr_setscope</code>



## Attribute Initialization and Deletion

---

Initialize or destroy an attribute with:

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



## Example: Create a detached thread

---

```
int error, fd;
pthread_attr_t tattr;
pthread_t tid;

if (error = pthread_attr_init(&tattr))
    fprintf(stderr, "Failed to create attribute object: %s\n",
            strerror(error));
else if (error = pthread_attr_setdetachstate(&tattr,
            PTHREAD_CREATE_DETACHED))
    fprintf(stderr, "Failed to set attribute state to detached: %s\n",
            strerror(error));
else if (error = pthread_create(&tid, &tattr, processfd, &fd))
    fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
```



## The thread stack

---

You can set a location and size for the thread stack.

```
int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                        void **restrict stackaddr, size_t *restrict stacksize);
int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
                        size_t stacksize);
```

Some systems allow you to set a guard for the stack so that an overflow into the guard area can generate a SIGSEGV signal.

```
int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                            size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr,
                            size_t guardsize);
```



# Thread scheduling

---

```
int pthread_attr_getscope(const pthread_attr_t *restrict attr,  
                          int *restrict contentionscope);
```

```
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

The contentionscope can be PTHREAD\_SCOPE\_PROCESS or PTHREAD\_SCOPE\_SYSTEM.

The scope determines whether the thread competes with other threads of the process or with other processes in the system.



## Create a thread that contends with other processes

---

```
int error;  
int fd;  
pthread_attr_t tattr;  
pthread_t tid;  
if (error = pthread_attr_init(&tattr))  
    fprintf(stderr, "Failed to create an attribute object:%s\n",  
            strerror(error));  
else if (error = pthread_attr_setscope(&tattr,  
                                       PTHREAD_SCOPE_SYSTEM))  
    fprintf(stderr, "Failed to set scope to system:%s\n",  
            strerror(error));  
else if (error = pthread_create(&tid, &tattr, processfd, &fd))  
    fprintf(stderr, "Failed to create a thread:%s\n",  
            strerror(error));
```



# Mutex Locking

---

- **Standard Initializer**

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

- **Static Initializer**

```
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;
```

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



## Example-I: Mutex Locking

---

```
pthread_mutex_t g_mutex =PTHREAD_MUTEX_INITIALIZER;
```

```
void func(){
```

```
...
```

```
pthread_mutex_lock(&g_mutex);
```

Atomic access to shared variable

```
pthread_mutex_unlock(&global_data_mutex)
```



## Example-I: Mutex Locking

---

```
pthread_mutex_t *g_mutex;

g_mutex=(pthread_mutex_t *)malloc(sizeof(pthread_mutex_t);
pthread_mutex_init(g_mutex, NULL);

void func(){
...
pthread_mutex_lock(&g_mutex);

        Atomic access to shared variable

pthread_mutex_unlock(&g_mutex)
```



## Mutex Locking

---

- **pthread\_mutex\_lock:**  
Blocks the calling thread until it is granted the lock. If the mutex is unlocked at the time of the call the lock is granted immediately; otherwise is granted after it's released by the thread that is holding it.
- **pthread\_mutex\_unlock:**  
Releases a lock. If a lock is not released, deadlock may occur.
- **pthread\_mutex\_trylock:**  
Locks a mutex. However, it does not block the caller if another thread has the mutex.



# Contention for a Mutex

---

- If more than one thread is waiting on a locked thread, which thread is the first to be granted the lock once it is released?
  - The thread with the highest priority gets the lock.



# Thread Safety

---

- count is a shared variable and needs to be protected from race conditions

- **Bad**

```
static int count;
void increment(void) {
    count++;
}
void decrement(void) {
    count--;
}
int getcount() {
    return count;
}
```

- **Good**

```
static int count = 0;
static pthread_mutex_t countlock =
    PTHREAD_MUTEX_INITIALIZER;

int increment(void) { /* decrement() similar */
    int error;
    if (error = pthread_mutex_lock(&countlock))
        return error;
    count++;
    return pthread_mutex_unlock(&countlock);
}

int getcount(int *countp) {
    int error;
    if (error = pthread_mutex_lock(&countlock))
        return error;
    *countp = count;
    return pthread_mutex_unlock(&countlock);
}
```



# Condition Variables

---

- A condition variable lets threads synchronize on the value of the data.
  - A thread waits until data reaches a particular state or until a certain event occurs.
  - Condition variables provide a kind of notification system among threads.



## Example: condition variables

---

```
int count = 0;
pthread_mutex_t count_mutex =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t count_threshold=
    PTHREAD_COND_INITIALIZER;
....
void watch_count() {
    pthread_mutex_lock(&count_mutex)
    while (count <= WATCH_COUNT) {
        pthread_cond_wait(&count_threshold,
            &count_mutex)
    }
    pthread_mutex_unlock(&count_mutex);
}
```

```
void inc_count(){
    for i=0; i< TCOUNT; i++){
        pthread_mutex_lock(&count_mutex);
        count ++
        if (count == WATCH_COUNT)
            pthread_cond_signal
                (&count_threshold);
        pthread_mutex_unlock(&count_mutex);
    }
}
```



# Condition Variables

---

- `pthread_cond_wait` & `pthread_cond_timedwait`  
Blocks the calling thread until another thread signals on the condition variable. `pthread_cond_timedwait` lets specify a timeout argument. If the condition is not signaled in the specified time, the thread is released from its wait.
- `pthread_cond_signal` & `pthread_cond_broadcast`  
Release threads that are waiting on a condition variable.  
`pthread_cond_signal` wakes up only one of the potentially many threads waiting on the condition; `pthread_cond_broadcast` awakens all of them.



## When many threads are waiting

---

- If multiple threads are waiting on a condition variable, who gets awakened first when another thread issues a `pthread_cond_signal`?
  - According to their priority.
  - If all the threads have the same priority, they are released in a first-in, first-out order for each `pthread_cond_signal` call that is issued.
- `pthread_cond_broadcast` releases all the threads waiting on the condition variable.
  - However, the system selects only one to give possession of the mutex.
  - The chosen thread is given the mutex and continues the execution of the code after `pthread_cond_wait`.
  - The other threads are moved to the queue of threads that are waiting to acquire the mutex. Each will resume as each previous thread in the queue acquires the mutex and then releases it.



# Spurious Wake Ups

---

- Look at the while loop in `watch_count()`. Is it really necessary?
  - Another thread could have awakened and decrease the counter, before our thread was able to execute.
  - We want to guard against a condition called spurious wake up. Perhaps a signaling thread has (in error, or due to an expected condition) awakened our thread when the expected condition has not in fact met.

The pthreads library allows an underlying threads library to issue spurious wake ups to a waiting thread without violating the standard. This while loop will guard against this possibility.



## Signals

---

- A special signal action structure (`sigaction`) allows a process to associate an action with each type of signal that may be delivered to it. A process may choose to:
  - Ignore the signal (`SIG_IGN`)
  - Use the default action (`SIG_DFL`). In most cases, this will end the process.
  - Catch the signal, and execute a user-specified handler.



## Signals in a Multithreaded World

---

- If multiple threads are executing within a process when a signal is delivered to it, the system must select a thread to process it.
- There are three possibilities



## Signals in a Multithreaded World

---

How the signal was generated	What generated the signal	Effective target of the signal	How the signal-processing thread is selected
Synchronously	The system, because of an exception	A specific thread	Always the offended thread
Synchronously	An internal thread using <code>pthread_kill</code>	A specific thread	Always the targeted thread
Asynchronously	An external process using <code>kill</code>	The process as a whole	Per-thread signal masks of all threads in the process



## Signals in a Multithreaded World

---

- Asynchronous signals:
  - If any thread can handle the signal: Unblock the signal in all the threads.
  - If only certain threads can handle the signal, unblock the signal only for those threads. The system will select one.
  - If only one thread can handle the signal, mask the signal in all other threads.



## Signals in a Multithreaded World

---

- Each thread has its own signal mask. However, all threads in a process share the signal action (sigaction) structure.
  - If a process specifies that a given signal should be ignored, it will be ignored regardless of to which thread in the process the system delivers it.
  - If the process's sigaction specifies that the signal should be subjected to the default action or processed by a signal handler, that action will be executed when the signal is delivered to any of the process's threads.



## Threads in signal handlers

---

- Pthreads labels calls that can be made safely from a signal handler as a asynchronous signal-safe functions. These are calls like alarm, getpid, mkdir, read , ...
- Pthreads calls cannot be safely done inside a signal handler (The standard specifies that the behaviour of Pthreads calls are undefined when the function is called from a signal handler).
  - What about access to shared memory?
    - Real-time extensions, can use sem-post.
    - Use sigwait.

