Threads: POSIX API 'Pthreads'
Creating a Thread

- When a new thread is created it runs concurrently with the creating thread.

- When creating a thread you indicate which function the thread should execute.
Compare: Normal function call (one thread)

Calling program:

```c
processfd();
```

Called function:

```c
processfd() {
}
```

Thread of execution
Compare: Threaded function call

Creating program

Calling program

Called function

processfd();

processfd() { 

created thread

processfd() {

Thread creation

Thread of execution
Threads vs. Processes

- **Process**
  - *fork* is expensive (time & memory)
  - each process has its own virtual addr. space

- **Thread**
  - Lightweight process
  - Shared virtual address space
  - Does not require lots of memory or startup time
Design choices: Processes versus Threads

Figure 4.1  Threads and Processes [ANDE97]
Copyright ©: University of Illinois CS 241 Staff
Thread-Specific Resources

- Each thread has its own
  - pthread_t identifier
  - Stack, Registers state, Program Counter

- Threads within the same process can communicate using shared memory
  - Must be done carefully!
  - Virtual memory is shared
Each process can include many threads

All threads of a process share:
- Process ID
- Virtual Memory (program code and global data)
- Open file/socket descriptors
- Semaphores
- Signal handlers
- Working environment (current directory, user ID, etc.)
Threads and address space

From: https://computing.llnl.gov/tutorials/pthreads/images/thread.gif
### Process Creation vs. Thread Creation

<table>
<thead>
<tr>
<th>Platform</th>
<th>fork()</th>
<th></th>
<th></th>
<th>pthread_create()</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>real</td>
<td>user</td>
<td>sys</td>
<td>real</td>
<td>user</td>
<td>sys</td>
</tr>
<tr>
<td>AMD 2.3 GHz Opteron (16 cpus)</td>
<td>12.5</td>
<td>1.0</td>
<td>12.5</td>
<td>1.2</td>
<td>0.2</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>AMD 2.4 GHz Opteron (8 cpus)</td>
<td>17.6</td>
<td>2.2</td>
<td>15.7</td>
<td>1.4</td>
<td>0.3</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>IBM 4.0 GHz POWER6 (8 cpus)</td>
<td>9.5</td>
<td>0.6</td>
<td>8.8</td>
<td>1.6</td>
<td>0.1</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>IBM 1.9 GHz POWER5 p5-575 (8 cpus)</td>
<td>64.2</td>
<td>30.7</td>
<td>27.6</td>
<td>1.7</td>
<td>0.6</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>IBM 1.5 GHz POWER4 (8 cpus)</td>
<td>104.5</td>
<td>48.6</td>
<td>47.2</td>
<td>2.1</td>
<td>1.0</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>INTEL 2.4 GHz Xeon (2 cpus)</td>
<td>54.9</td>
<td>1.5</td>
<td>20.8</td>
<td>1.6</td>
<td>0.7</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td>INTEL 1.4 GHz Itanium2 (4 cpus)</td>
<td>54.5</td>
<td>1.1</td>
<td>22.2</td>
<td>2.0</td>
<td>1.2</td>
<td>0.6</td>
<td></td>
</tr>
</tbody>
</table>

- Timings reflect 50,000 process/thread.
- Creations were performed with the time utility, and units are in seconds, no optimization flags.

Copyright ©: University of Illinois CS 241 Staff
POSIX and threads

- Early on
  - Each OS had its own thread library/API
  - Difficult to write multithreaded programs
    - Learn a new API with each new OS
    - Modify code with each port to a new OS

- So
  - POSIX (IEEE 1003.1c-1995) provided a standard known as pthreads
## Pthread Operations

<table>
<thead>
<tr>
<th>POSIX function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create</td>
<td>create a thread</td>
</tr>
<tr>
<td>pthread_detach</td>
<td>set thread to release resources</td>
</tr>
<tr>
<td>pthread_equal</td>
<td>test two thread IDs for equality</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>exit a thread without exiting process</td>
</tr>
<tr>
<td>pthread_kill</td>
<td>send a signal to a thread</td>
</tr>
<tr>
<td>pthread_join</td>
<td>wait for a thread</td>
</tr>
<tr>
<td>pthread_self</td>
<td>find out own thread ID</td>
</tr>
</tbody>
</table>
Creating a Thread

```c
int pthread_create (pthread_t* tid, pthread_attr_t* attr, void *(child_main)(void*), void* arg);
```

- **creates a new posix thread**
- **Parameters:**
  - `tid`:
    - Unique thread identifier returned from call
  - `attr`:
    - Attributes structure used to define new thread
    - Use **NULL** for default values
  - `child_main`:
    - Main routine for child thread
    - Takes a pointer (void*), returns a pointer (void*)
  - `arg`:
    - Argument pointer passed to child thread

Copyright ©: University of Illinois CS 241 Staff
Creating a Thread

- `pthread_create()` takes a pointer to a function as one of its arguments
  - `child_main` is called with the argument specified by `arg`
  - `child_main` can only have one parameter of type `void *`
  - Complex parameters can be passed by creating a structure and passing the address of the structure
  - **The structure can't be a local variable**
  - By default, a new thread is created in a **joinable state**

- **Thread ID**
  - `pthread_t pthread_self(void);`
  - Returns ID of executing thread
Exiting a thread

Question:
- If a thread calls exit(), what about other threads in the same process?

When does a multithreaded process terminate?
Exiting a thread

Question:
- If a thread calls exit(), what about other threads in the same process?

A multithreaded process terminates when:
- one of its threads calls exit
- it returns from main()
- it receives a termination signal
- all threads have called pthread_exit

In any of these cases, all threads of the process terminate.
Terminating Threads: `pthread_exit()`

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

- Terminate the calling thread
- Makes the value `retval` available to any successful join with the terminating thread
- Returns
  - `pthread_exit()` cannot return to its caller
- Parameters
  - `retval`:
    - Pointer to data returned to joining thread
    - *Pass a pointer to heap not to the stack*
- **Note**
  - If `main()` exits by calling `pthread_exit()` before its threads, the other threads continue to execute. Otherwise, they will be terminated when `main()` finishes.

Copyright ©: University of Illinois CS 241 Staff
Detaching Threads:

`pthread_detach()`

```c
int pthread_detach(pthread_t thread);
```

- Thread resources can be reclaimed on termination
- Return results of a detached thread are unneeded
- Returns
  - 0 on success
  - Error code on failure
- Parameters
  - `thread`:
    - Target thread identifier
- Notes
  - `pthread_detach()` can be used to explicitly detach a thread even though it was created as joinable
  - There is no converse routine

Detached Threads

Master Thread

Worker Thread

pthread_create()

Worker Thread

Worker Thread

pthread_create()

Worker Thread

pthread_exit()

Worker Thread

pthread_exit()

Worker Thread

pthread_exit()
Waiting for Threads: 

`pthread_join()`

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

- Suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.

- Returns
  - 0 on success
  - Error code on failure

- Parameters
  - `thread`:
    - Target thread identifier
  - `retval`:
    - The pointer passed to `pthread_exit()` by the terminating thread is made available in the location referenced by `retval`
Joined Threads

Master Thread

Worker Thread

suspends calling thread, retrieves void* retval

pthread_create()

pthread_join()

pthread_exit()
Example 1

```c
int x = 0;
char *p;
void *thread(void *th){
    x = x + 10;
    strcat(p, "Hello from thread!");
    printf("thread: my x is %d. Bye from thread!\n", x);
    pthread_exit((void *) p+5);
}

int main() {
    pthread_t tid;
    char *p_char;
    p_char = p = malloc(25 * sizeof(char));          // data allocated on heap
    strcpy (p, "main-thread: ");
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, (void **) &p_char);

    printf("%s\n", p_char);
    printf("main: my x is %d; Bye from main!\n", x);
}
```

Necessary includes:
```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>
```
Example 2

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

int x = 0;
char *p;
void *thread(void *th){
    x = x + 10;
    strcat(p, "Hello from thread!");
    printf("thread: my x is %d. Bye from thread!\n", x);
    pthread_exit((void *) p+5);
}

int main()
```
Valid outputs for example 2 (non-deterministic)

Output #1
---------------------------------------
main-thread: Hello from thread!
thread: my x is 10. Bye from thread!
main: my x is 10; Bye from main!

Output #2
---------------------------------------
main-thread:
thread: my x is 10. Bye from thread!
main: my x is 10; Bye from main!

Output #3
---------------------------------------
main-thread:
main: my x is 10; Bye from main!

Copyright ©: University of Illinois CS 241 Staff
pthread Error Handling

- pthread functions do not follow the usual Unix conventions
  - Similarity
    - Returns 0 on success
  - Differences
    - Returns error code on failure
    - Does not set `errno`
  - What about `errno`?
    - Each thread has its own
    - `errno is thread-local`; setting it in one thread does not affect its value in any other thread.
Threads vs processes

- Threads are similar to concurrent processes
  - **Pros**: thread creation is faster; data sharing among threads is fast and easy
  - **Cons**: application is less robust; data sharing requires synchronization to avoid race conditions

- If a thread misbehaves, it can corrupt data of other threads within same process

- If a thread crashes, the entire process crashes
# Threads vs. Processes

<table>
<thead>
<tr>
<th>Property</th>
<th>Processes created with fork</th>
<th>Threads of a process</th>
<th>Ordinary function calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables</td>
<td>Get copies of all variables</td>
<td>Share global variables</td>
<td>Share global variables</td>
</tr>
<tr>
<td>IDs</td>
<td>Get new process IDs</td>
<td>Share the same process ID</td>
<td>Share the same process ID (and thread ID)</td>
</tr>
<tr>
<td>Data/control</td>
<td>Must communicate explicitly, e.g., use pipes, shared memory, msg. passing.</td>
<td>May communicate with return value or carefully shared variables</td>
<td>May communicate with return value or shared variables</td>
</tr>
<tr>
<td>Parallelism (one CPU)</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Sequential</td>
</tr>
<tr>
<td>Parallelism (multiple CPUs)</td>
<td>May be executed simultaneously</td>
<td>May be executed simultaneously</td>
<td>Sequential</td>
</tr>
</tbody>
</table>