

## Thread Magic

How one process can do two things at once

- Thread of execution?
- Share process memory but each has its own call-stack

Create, Wait, Destroy

- How to use the POSIX API 'PThreads'

Threads and Processes

- When multi-threaded processes die

Copyright © Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

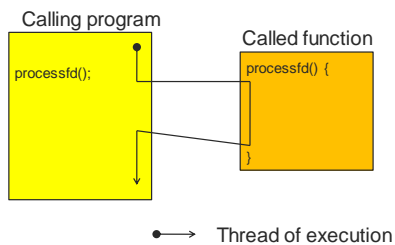
## Thread of Execution

- Sequential set of instructions
  - Function calls & automatic (local) variables
  - Need Program Counter and Stack for each thread

Copyright © Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta



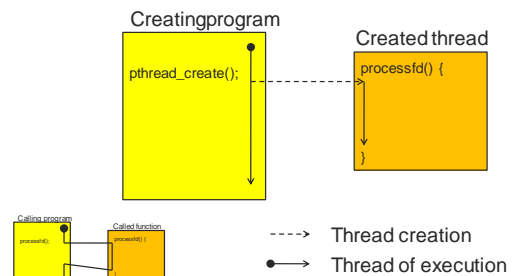
## Compare: Normal function call (1 thread)



Copyright © Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta



## Compare: Threaded function call



Copyright © Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta



## [ Threads vs. Processes ]

- Process
  - using fork is expensive (time & memory)
- Thread
  - Lightweight process
  - Does not require lots of memory or startup time

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



## [ Process and Threads ]

- Each process can include many threads
- All threads of a process share:
  - Process ID
  - Memory (program code and global data)
  - Open file/socket descriptors
  - Semaphores
  - Signal handlers and signal dispositions
  - Working environment (current directory, user ID, etc.)

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



## [ Thread-Specific Resources ]

- Each thread has it's own:
  - Thread ID (integer)
  - Stack, Registers, Program Counter
- Threads within the same process can communicate using shared memory.
  - Must be done carefully!

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



## [ The Pthreads API ]

- Thread management
  - Creating, detaching, joining, etc.
  - Set/query thread attributes
- Mutexes
  - Synchronization
- Condition variables
  - Communications between threads that share a mutex

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



## Creating a Thread

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

- Spawn 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 passed to child thread

Copyright © Nahstedt, Angrave, Abdebaizer, Kravets, Gupta



## 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
- Thread ID
  - **pthread\_t pthread\_self(void);**
  - Returns currently executing thread's ID

Copyright © Nahstedt, Angrave, Abdebaizer, Kravets, Gupta



## Example: pthread\_create()

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void *snow(void *data) {
    printf("Let it snow ... %s\n", data);
    pthread_exit(NULL);
}
int main(int argc, char *argv[]) {
    pthread_t mythread;
    int result;
    char *data = "Let it snow.";
    result = pthread_create(&mythread, NULL, snow, data);
    printf("pthread_create() returned %d\n", result);
    if(result)
        exit (1);
    pthread_exit(NULL);
}
```

Copyright © Nahstedt, Angrave, Abdebaizer, Kravets, Gupta



## pthread's Attributes

- Attributes
  - Data structure **pthread\_attr\_t**
  - Set of choices for a thread
  - Passed in thread creation routine
- Choices
  - Scheduling options (more later on scheduling)
  - Detached state
    - Detached
      - Main thread does not wait for the child threads to terminate
    - Joinable
      - Main thread waits for the child thread to terminate
      - Useful if child thread returns a value

Copyright © Nahstedt, Angrave, Abdebaizer, Kravets, Gupta



## pthread Attributes

- Initialize an attributes structure to the default values
  - `int pthread_attr_init (pthread_attr_t* attr);`
- Set the detached state value in an attributes structure
  - `int pthread_attr_setdetachedstate (pthread_attr_t* attr, int value);`
  - Value
    - `PTHREAD_CREATE_DETACHED`
    - `PTHREAD_CREATE_JOINABLE`

Copyright © Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta



## Detaching Threads: `pthread_detach()`

- ```
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

Copyright © Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta



## Waiting for Threads: `pthread_join()`

- ```
int pthread_join(pthread_t thread, void** retval);
```
- Suspend calling thread until target thread terminates
  - Returns
    - 0 on success
    - Error code on failure
  - Parameters
    - `thread`:
      - Target thread identifier
    - `retval`:
      - The value passed to `pthread_exit()` by the terminating thread is made available in the location referenced by `retval`.

Copyright © Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta



## Waiting for Threads: `pthread_join()`

- ```
int pthread_join(pthread_t thread, void** retval);
```
- Note
    - You cannot join on a detached thread.
    - Detaching means you are NOT interested in knowing about the thread's exit
  - Set `pthread_attr` to joinable when calling `pthread_create()`

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);
```

Copyright © Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta



## Terminating Threads: `pthread_exit()`

```
int 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
- Note
  - If `main()` exits before its threads, and exits with `pthread_exit()`, the other threads continue to execute. Otherwise, they will be terminated when `main()` finishes.

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



## Returning data through `pthread_join()`

```
void *thread(void *vargp) {
    pthread_exit((void *)42);
}

int main() {
    int i;
    pthread_t tid;

    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, (void **)&i);
    printf("%d\n", i);
}
```

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



## `pthread` Error Handling

- pthreads 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
    - Define `REENTRANT` (`-D_REENTRANT` switch to compiler) when using pthreads

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



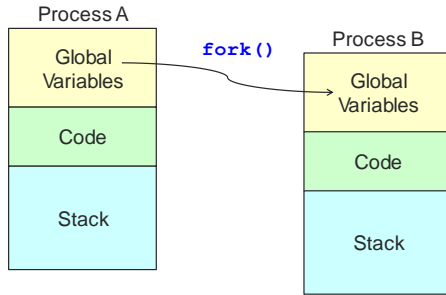
## Thread vs. Process Creation

- `fork()` clones the process
  - Two separate processes with independent destinies
  - Independent memory space for each process
- `pthread_create()`
  - Start from a function
  - Share memory

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



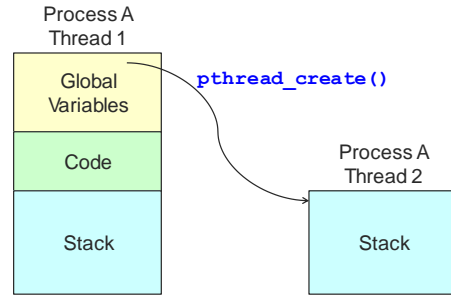
## [ fork () ]



Copyright © Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta



## [ pthread\_create () ]



Copyright © Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta



## [ Possible output? ]

```
int x = 1;
fork();
x = x+1;
printf("x is %d\n");
```

Copyright © Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta



## [ Possible output? ]

```
int x = 1;
main(...) {
    pthread_t tid;
    pthread_create(
        &tid, NULL,
        func, NULL);
    func(NULL);
}

void* func(void*p) {
    x = x + 1;
    printf("x is
        %d\n");
    return NULL;
}
```

Copyright © Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta



## Thread Lifetime

- A thread exists until
  - It returns from the function or calls `pthread_exit()`
  - The whole process terminates
  - The machine catches fire

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



## So, your process terminates when...

1. Any thread calls `exit()`;
2. The main thread returns `main() { pthread_create(); return 0; }`
3. Segmentation fault `*(char*)0 = 0;`
4. There are no more threads left to run

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



## Main points

- When creating a thread you indicate which function the thread should execute
- When a new thread is created it runs concurrently with the creating thread
- A thread is the lightest unit of work that can be scheduled to run on the processor

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



## Why Use Threads Over Processes?

- Creating a new process can be expensive
  - Time
    - A call into the operating system is needed
    - Context-switching involves the operating system
  - Memory
    - The entire process must be replicated
  - The cost of inter-process communication and synchronization of shared data
    - May involve calls into the operation system kernel
- Threads can be created without replicating an entire process
  - Creating a thread is done in user space rather than kernel

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



## Threads vs. Processes

| 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)             |
| Data/control                | Must communicate explicitly, e.g., use pipes or small integer return value | May communicate with return value or carefully shared variables | May communicate with return value or shared variables |
| Parallelism (one CPU)       | Concurrent                                                                 | Concurrent                                                      | Sequential                                            |
| Parallelism (multiple CPUs) | May be executed simultaneously                                             | Kernel threads may be executed simultaneously                   | Sequential                                            |

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta



## Take-away questions

- Why are threads useful?
  - Why not just create concurrent processes?
- What support is needed by the O/S?
- What could happen if a thread makes a blocking I/O call?

Copyright © Nahstedt, Angrave, Abdelzaher, Kravets, Gupta

