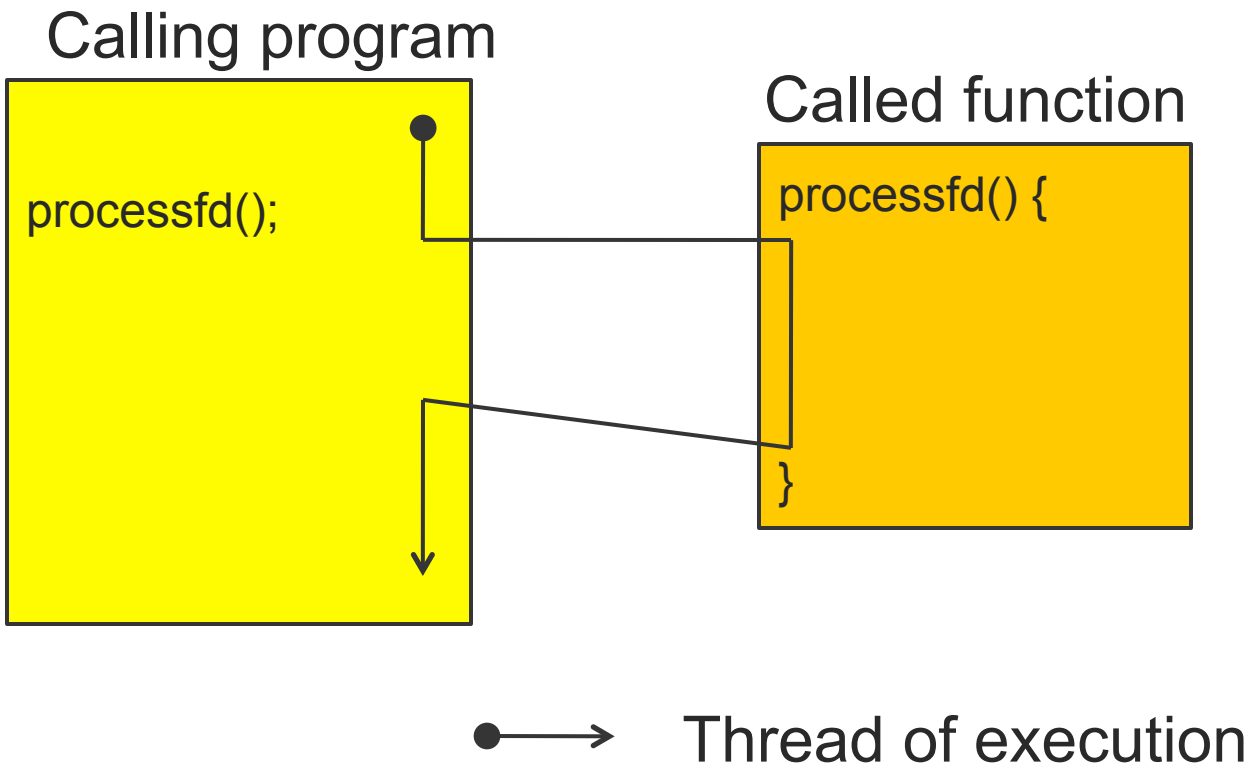
Threads:
POSIX API 'Pthreads'

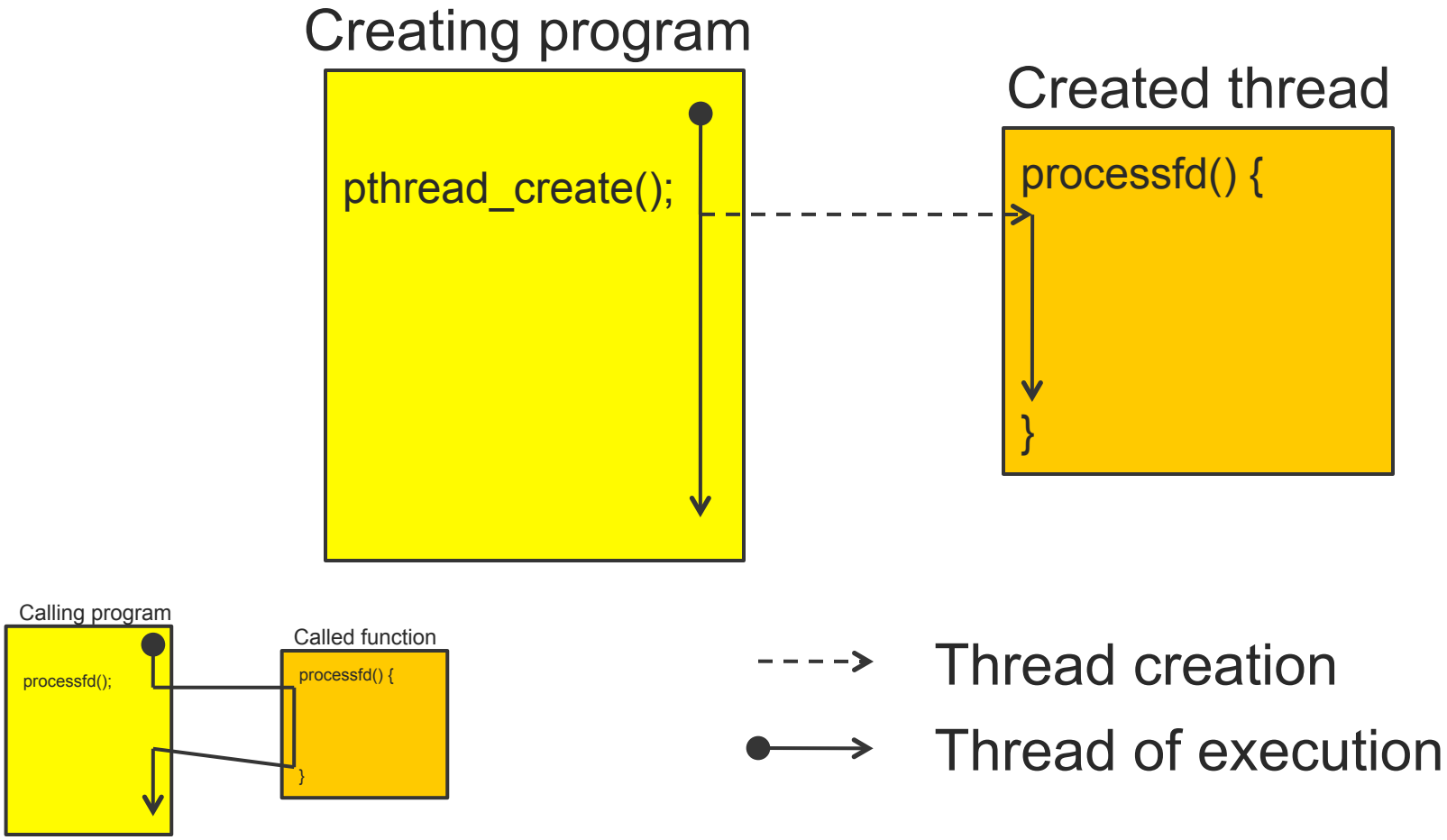
Creating a Thread

- 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)



Compare: Threaded function call



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

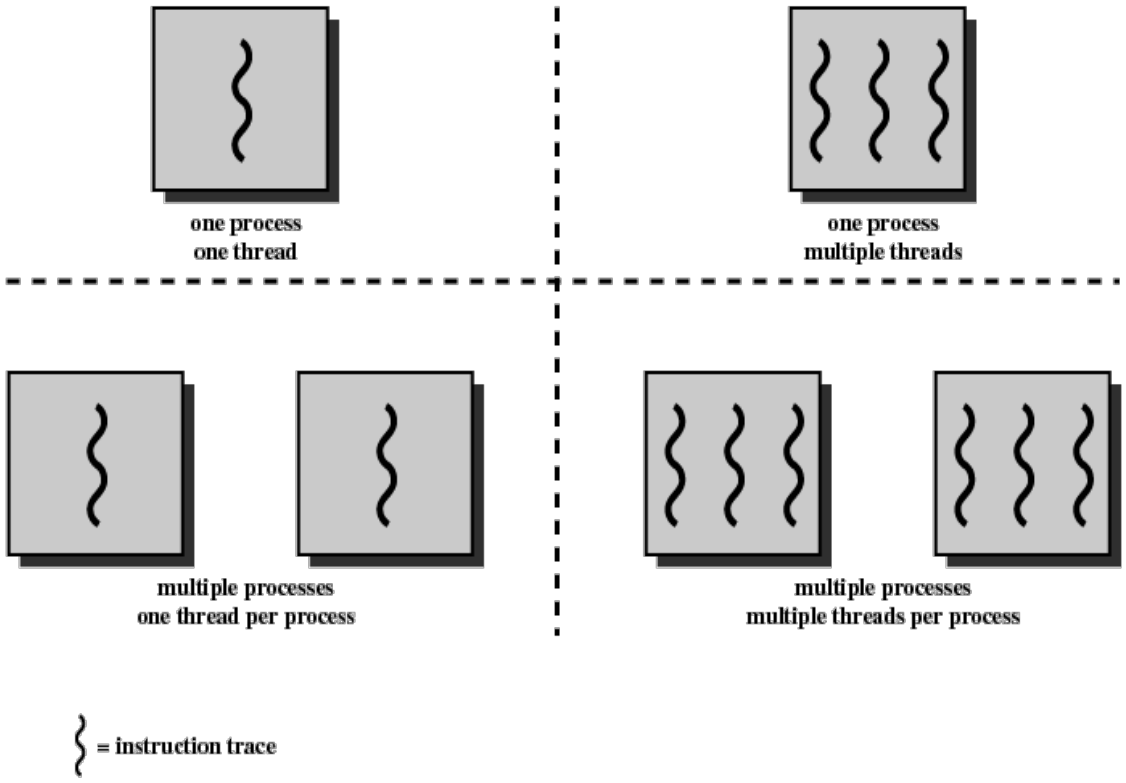
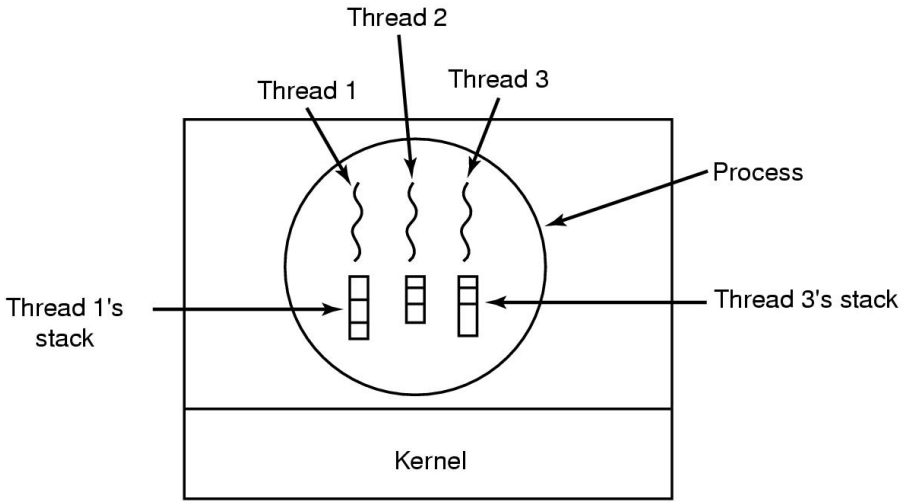


Figure 4.1 Threads and Processes [ANDE97]

Thread-Specific Resources

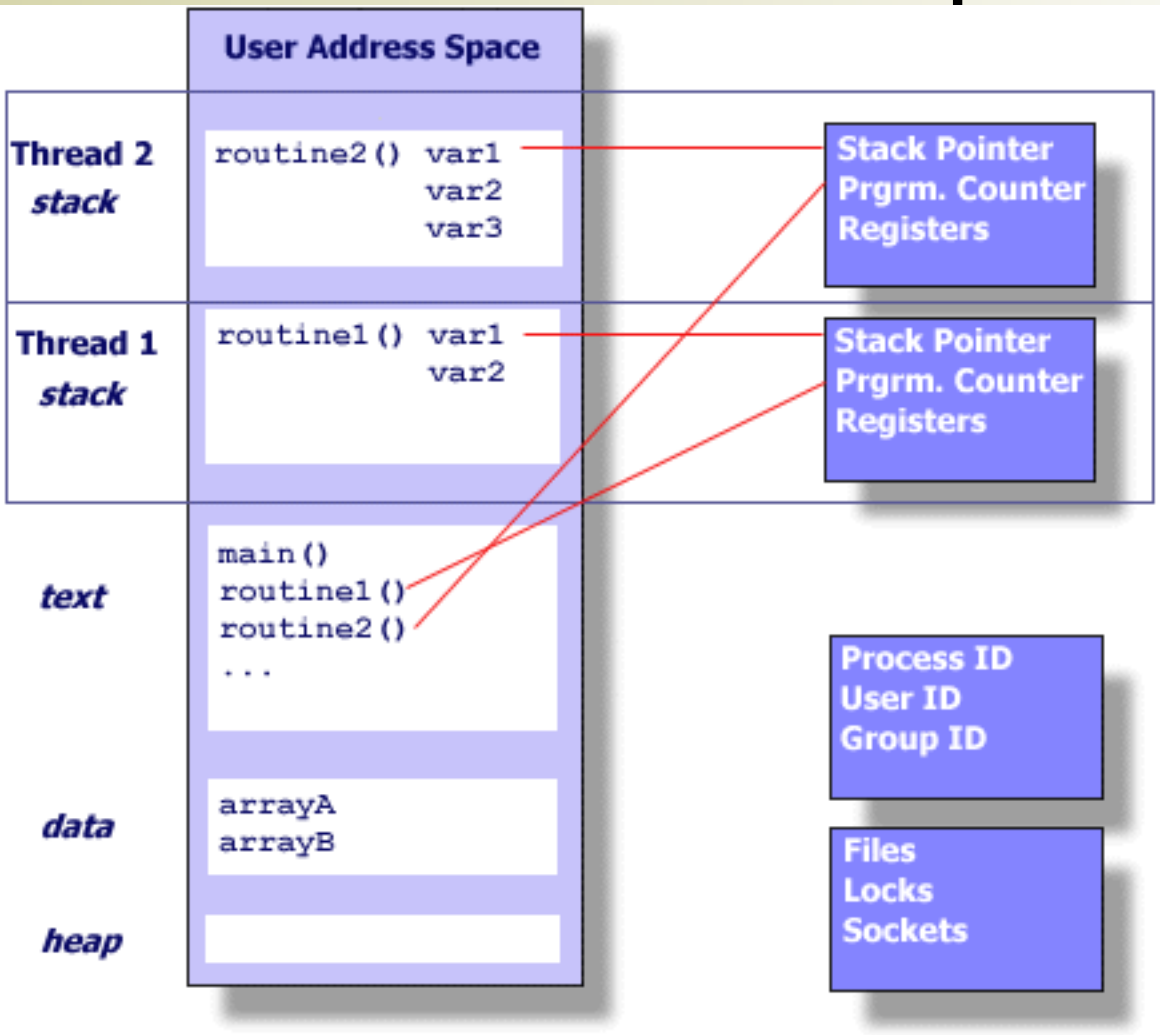
- Each thread has its own
  - pthread_t identifier
  - Stack, Registers state, Program Counter
- Threads within the same process can communicate using shared memory
  - Must be done carefully!
  - Virtual memory is shared



Process and Threads

- 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

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

- http://www.llnl.gov/computing/tutorials/pthreads.
- Timings reflect 50,000 process/thread.
- Creations, were performed with the time utility, and units are in seconds, no optimization flags.

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

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

Creating a Thread

```
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

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()`

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

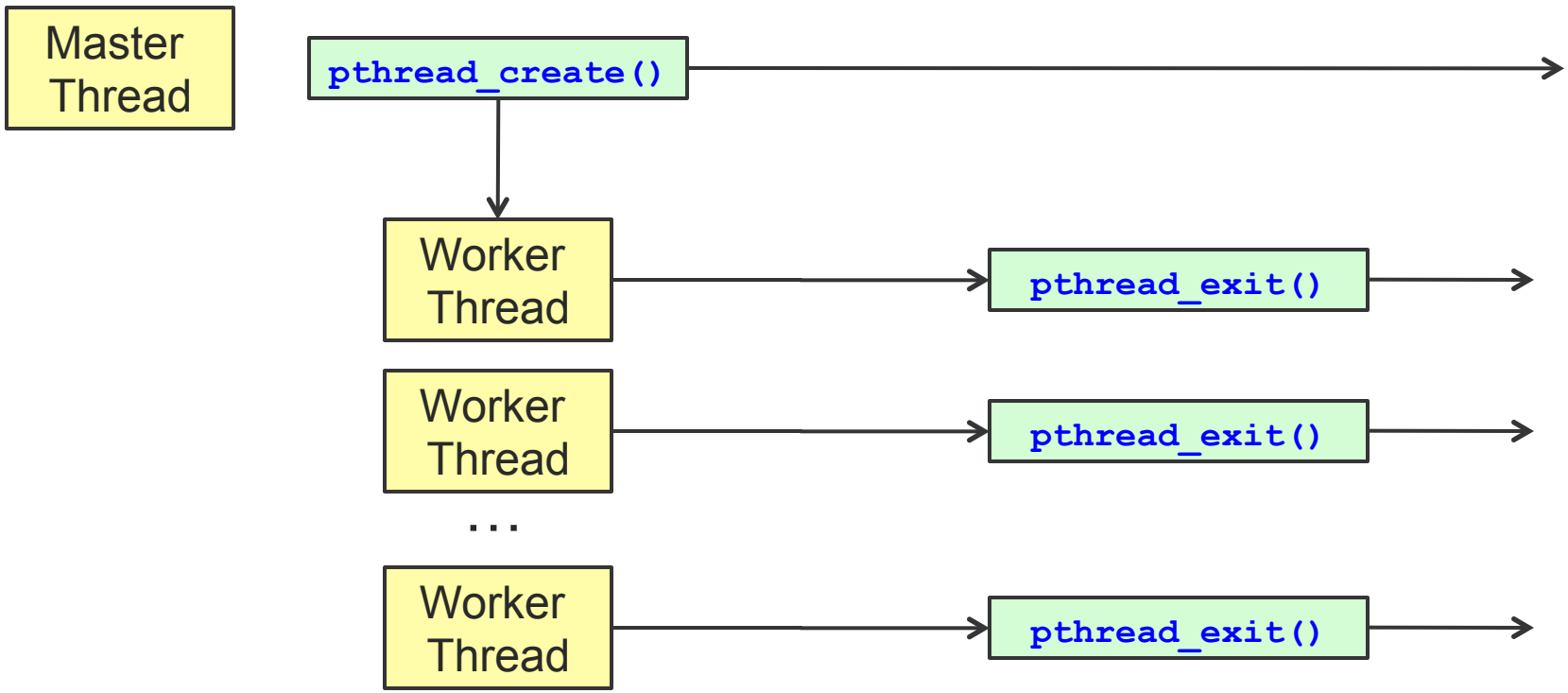
- Terminate the calling thread
- Makes the value **`retval`** available to any successful join with the terminating thread
- Returns
  - **`pthread_exit()`** cannot return to its caller
- Parameters
  - **`retval`**:
    - Pointer to data returned to joining thread
    - *Pass a pointer to heap not to 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.

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
- 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



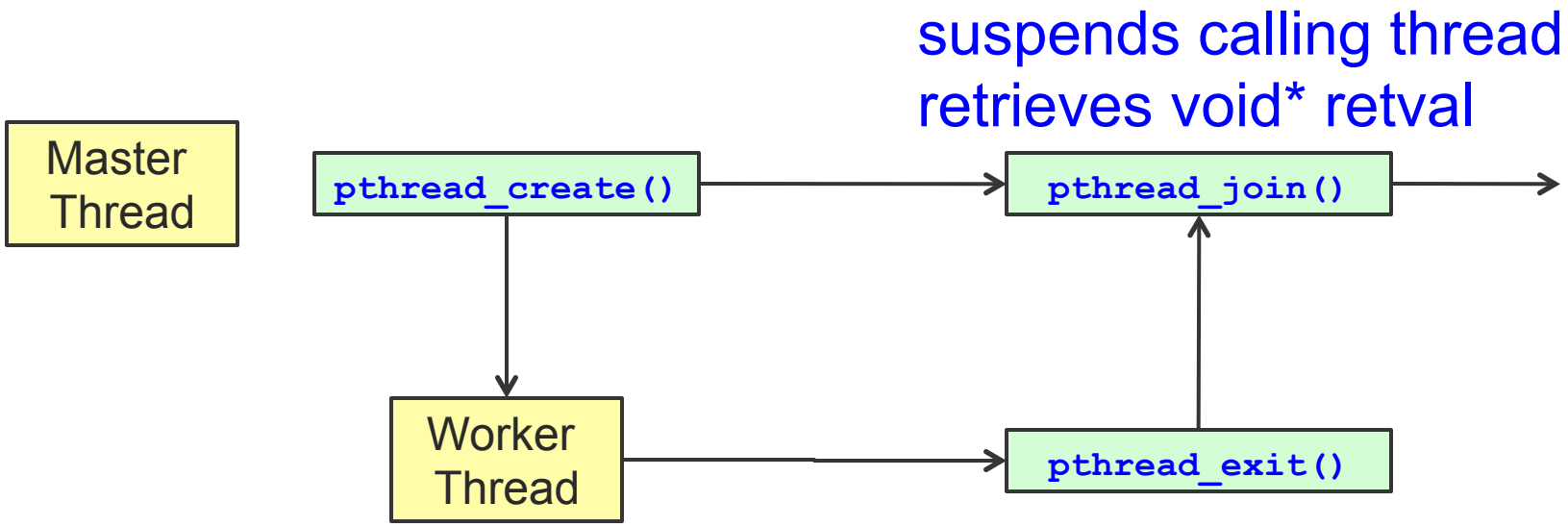
Waiting for Threads: `pthread_join()`

```
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



Example 1

```
Necessary includes:
#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() {
  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);
}
```


Example 2

```
Necessary includes:
#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() {
  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);
}
```

What is the output?

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!
```


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

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, shared memory, msg. passing.	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	May be executed simultaneously	Sequential