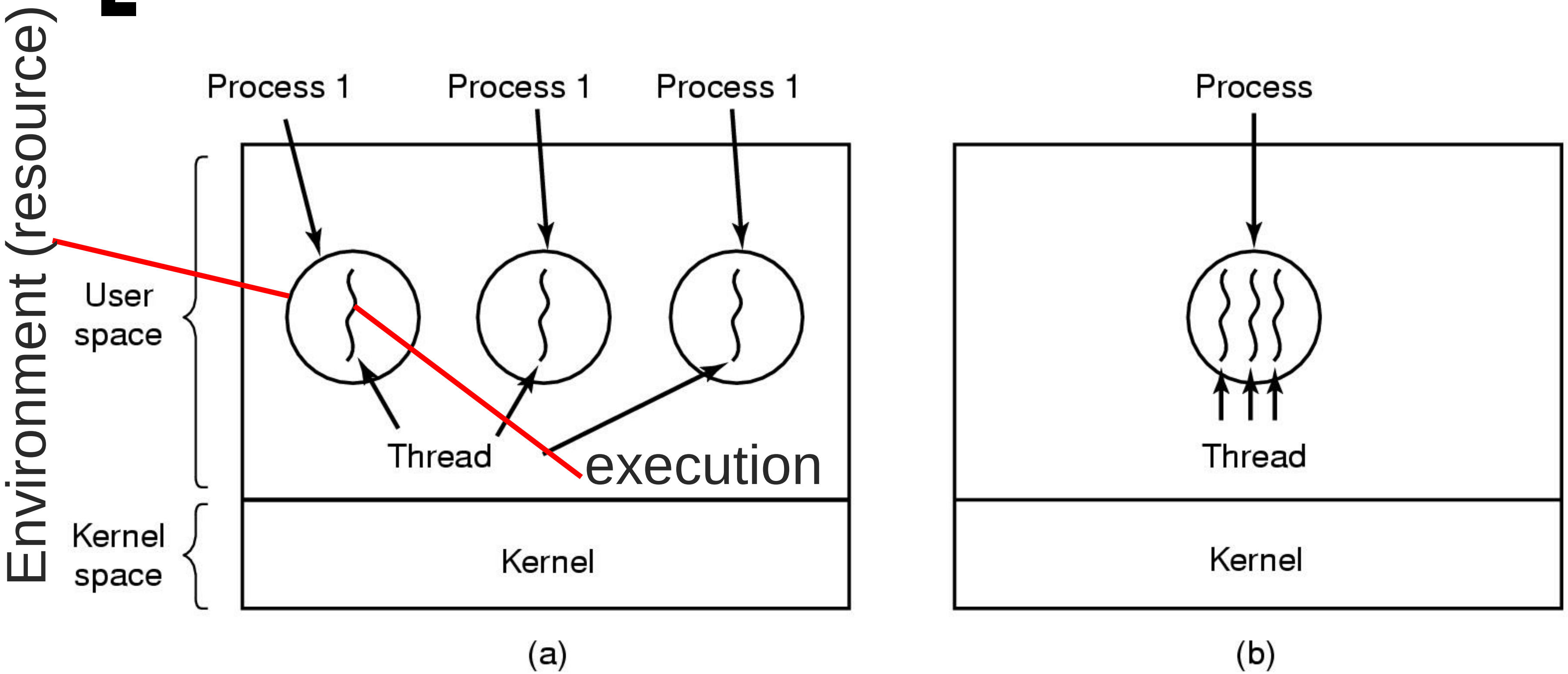
Thread Magic

- 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

Threads vs. Processes

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

Processes vs. Threads



a) Three processes each with one thread
b) One process with three threads

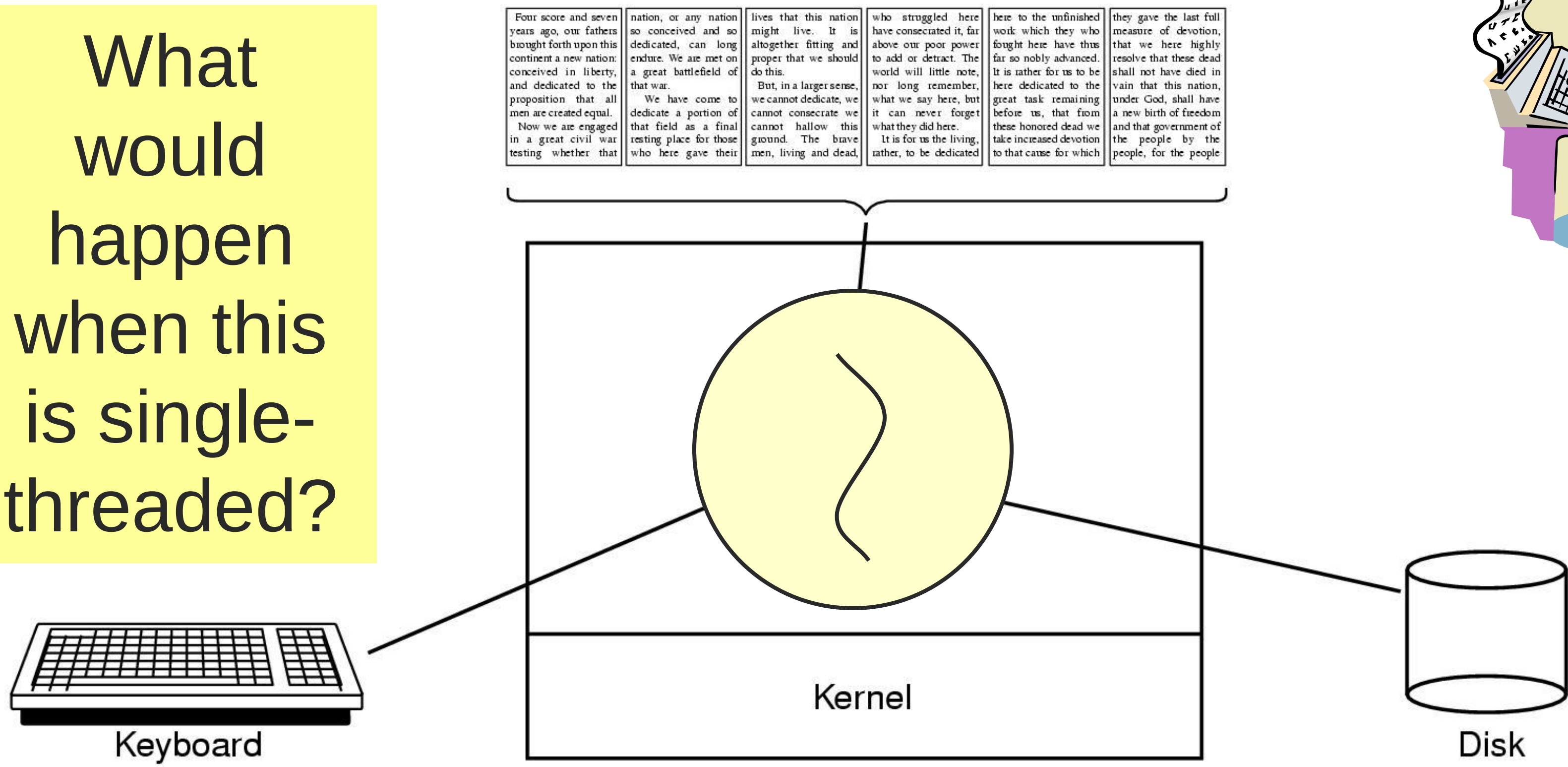
Process and Threads

- 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.)

Thread Usage: Word Processor

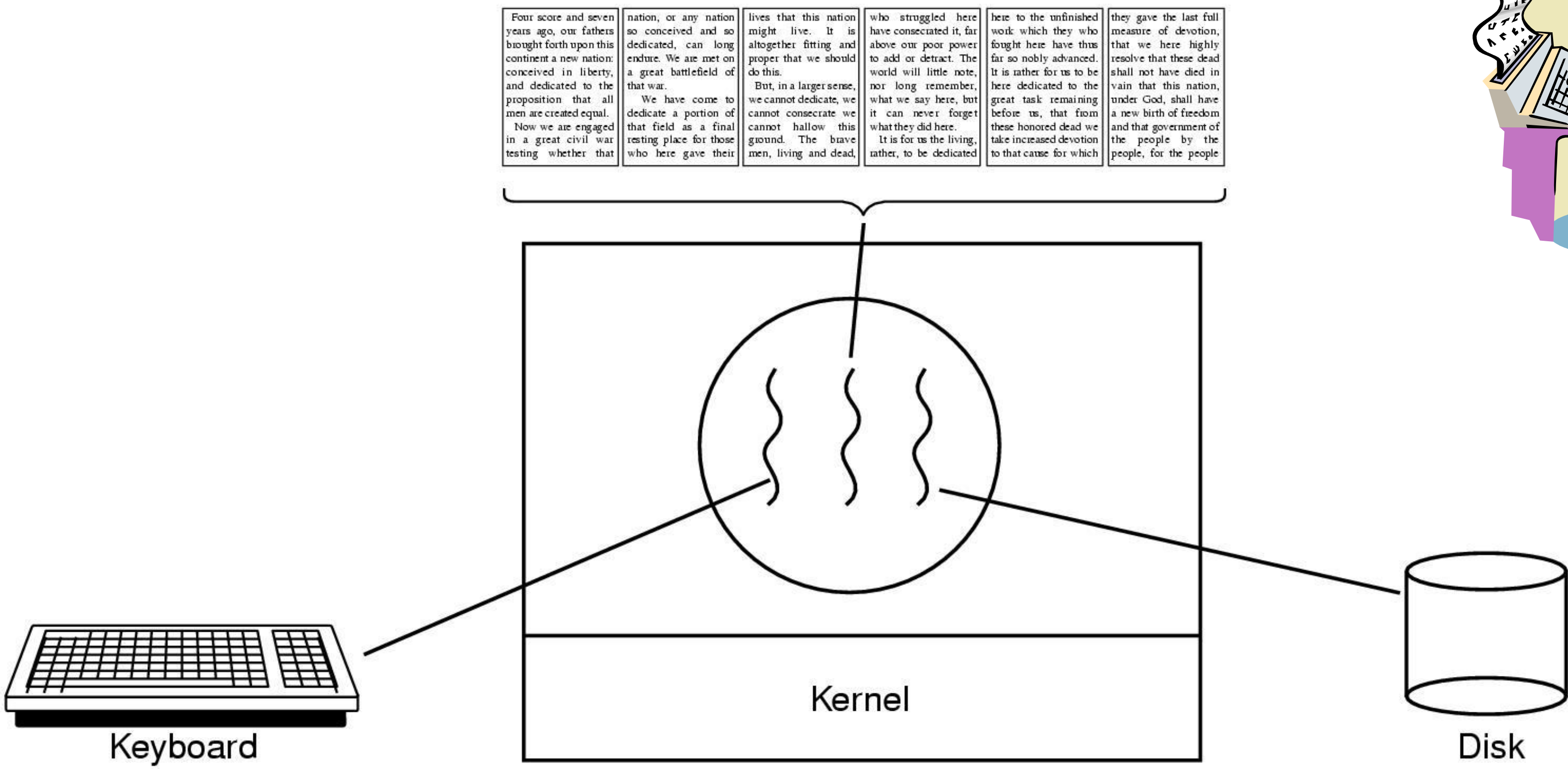
- Working file can only be accessed by one process at a time

What would happen when this is single-threaded?

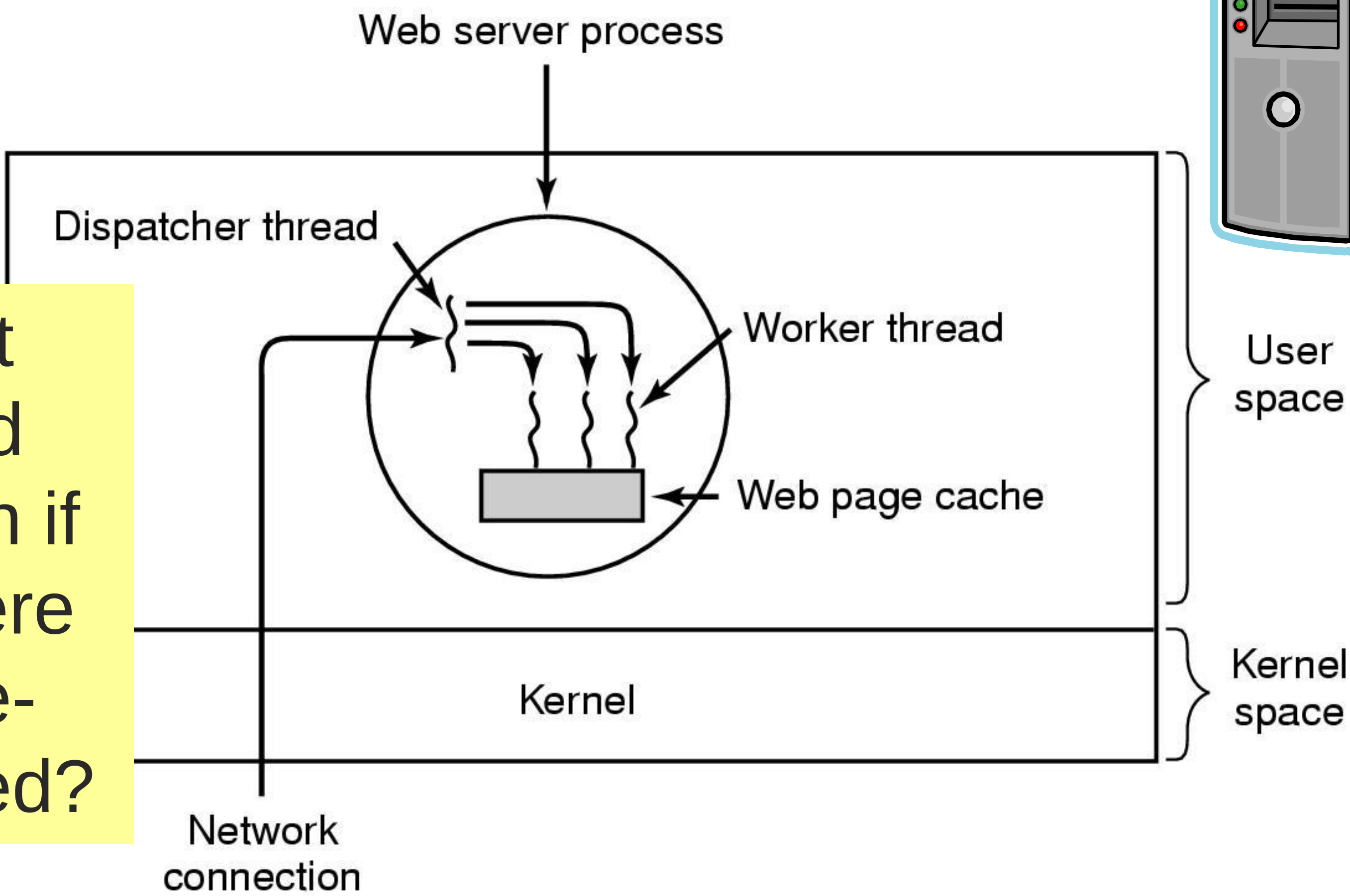


Thread Usage: Word Processor

- Working file can only be accessed by one process at a time



Thread Usage: Web Server



What would happen if this were single-threaded?

Web Server

- Pseudo-code for previous slide
  - Dispatcher thread

```
while (TRUE) {
 get_next_request(&buf);
   handoff_work(&buf);
}
```

  - Worker thread

```
while (TRUE) {
   wait_for_work(&buf);
   look_for_page_in_cache(&buf, &page);
   if (page_not_in_cache( &page))
   read_page_from_disk(&buf, &page);
   return_page(&page);
}
```

- Alternative
  - Dispatcher thread

```
while (TRUE) {
 get_next_request(&buf);
   handoff_work(&buf);
}
```

  - Worker thread

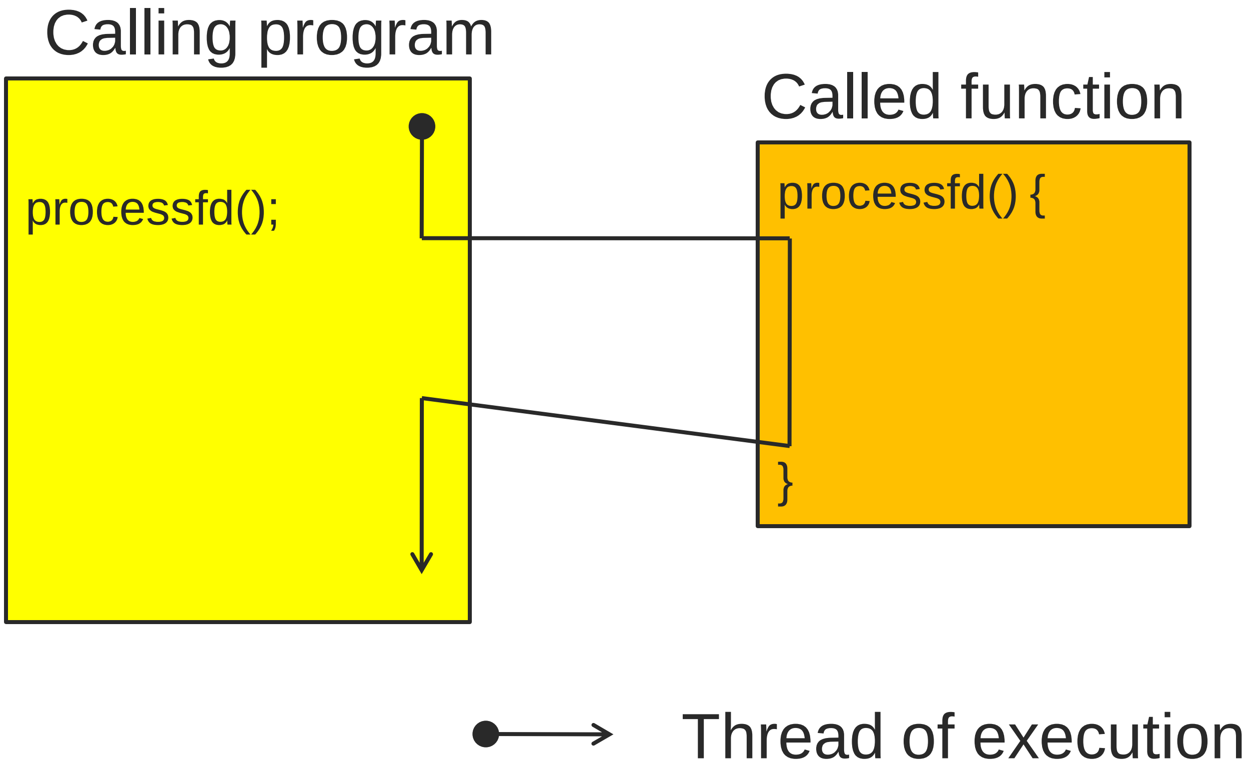
```
work (&buf) {
   look_for_page_in_cache(&buf, &page);
   if (page_not_in_cache( &page))
   read_page_from_disk(&buf, &page);
   return_page(&page);
}
```

What is the difference?

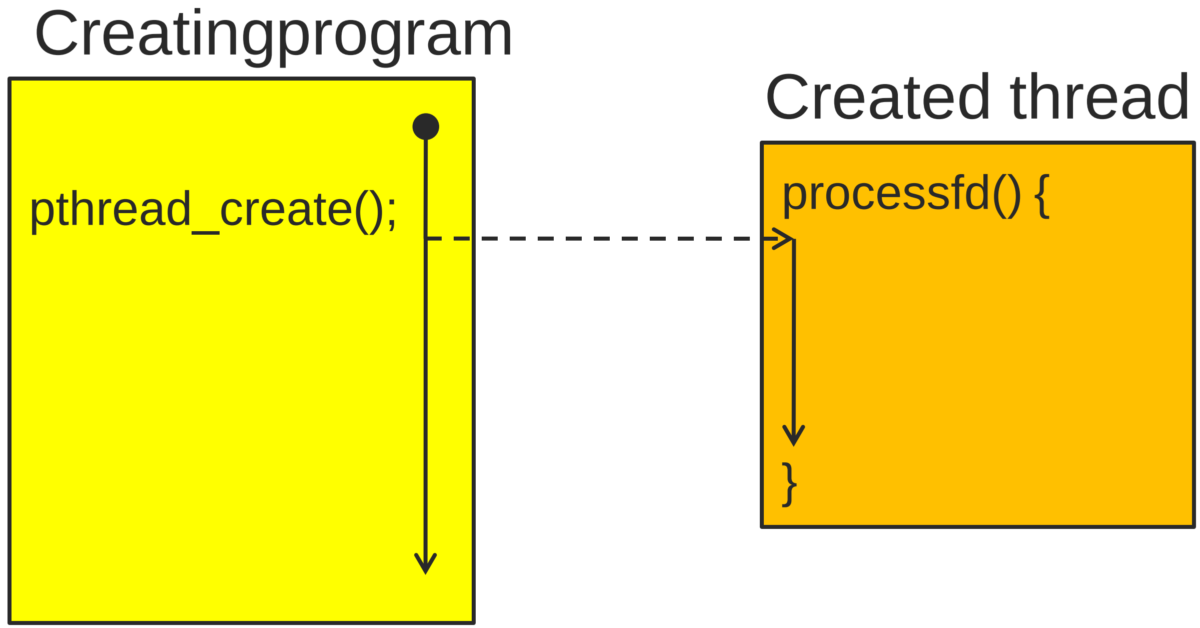
Thread of Execution

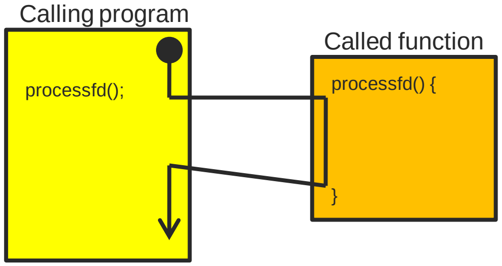
- Sequential set of instructions
  - Function calls & automatic (local) variables
  - Need Program Counter and Stack for each thread

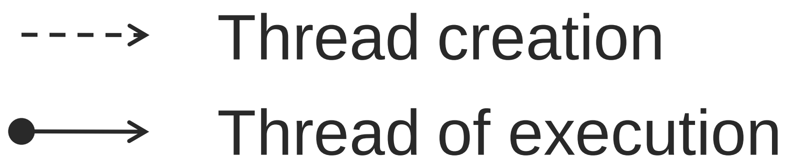
Compare: Normal function call (1 thread)



Compare: Threaded function call





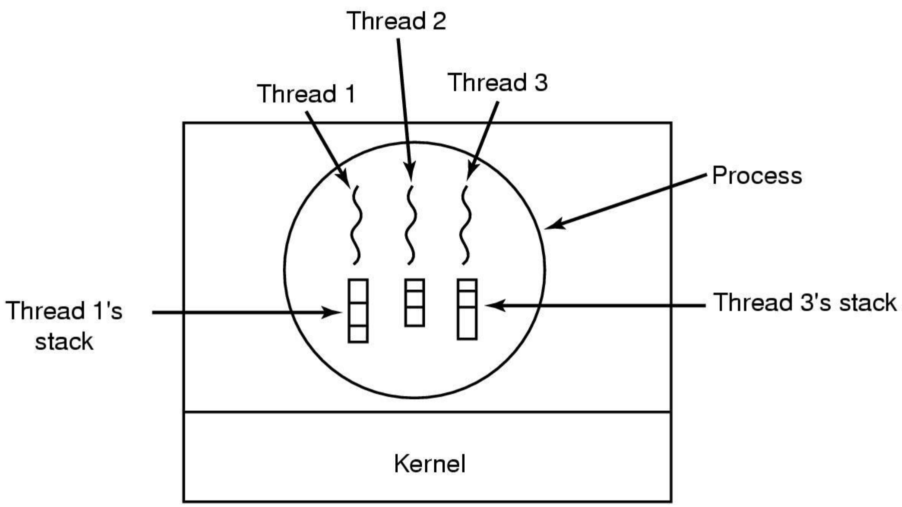


Thread Execution States

- States associated with a change in thread state
  - Spawn (another thread)
  - Block
    - Does blocking a thread block other, or all, threads
  - Unblock
  - Finish (thread)
    - De-allocate register context and stacks

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!

Processes vs. Threads

Per Process Items	Per Thread Items
Address space Global variables Open files Child processes Pending alarms Signals and signal handlers Accounting information	Program counter Registers Stack State

- Each thread executes separately
- Threads in the same process share many resources
- No protection among threads!! (What?)

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.

What's POSIX Got To Do With It?

- Early on
  - Each OS had it's 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

The pthreads API

Today

- Thread management
  - Creating, detaching, joining, etc. Set/query thread attributes

Next week

- Mutexes
  - Synchronization
- Condition variables
  - Communications between threads that share a mutex

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

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

Example: `pthread_create()`

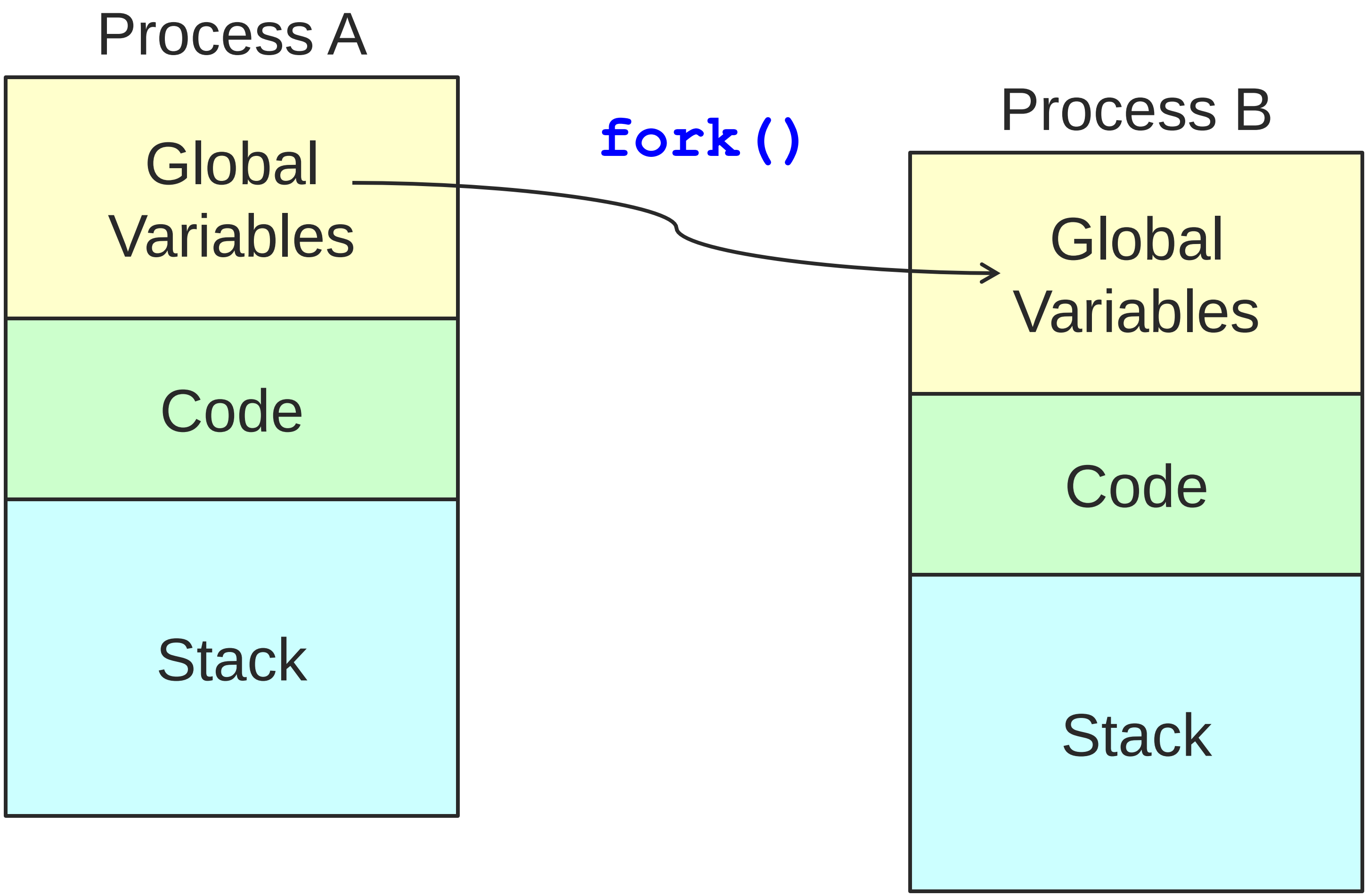
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void *snow(void *data) {
    printf("Let it snow ... %s\n", data);
    pthread_exit(NULL);
}
int main(int argc, char *argv[]) {
    pthread_t mythread;
    int result;
    char *data = "Let it snow.";
    result = pthread_create(&mythread,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

What is this?

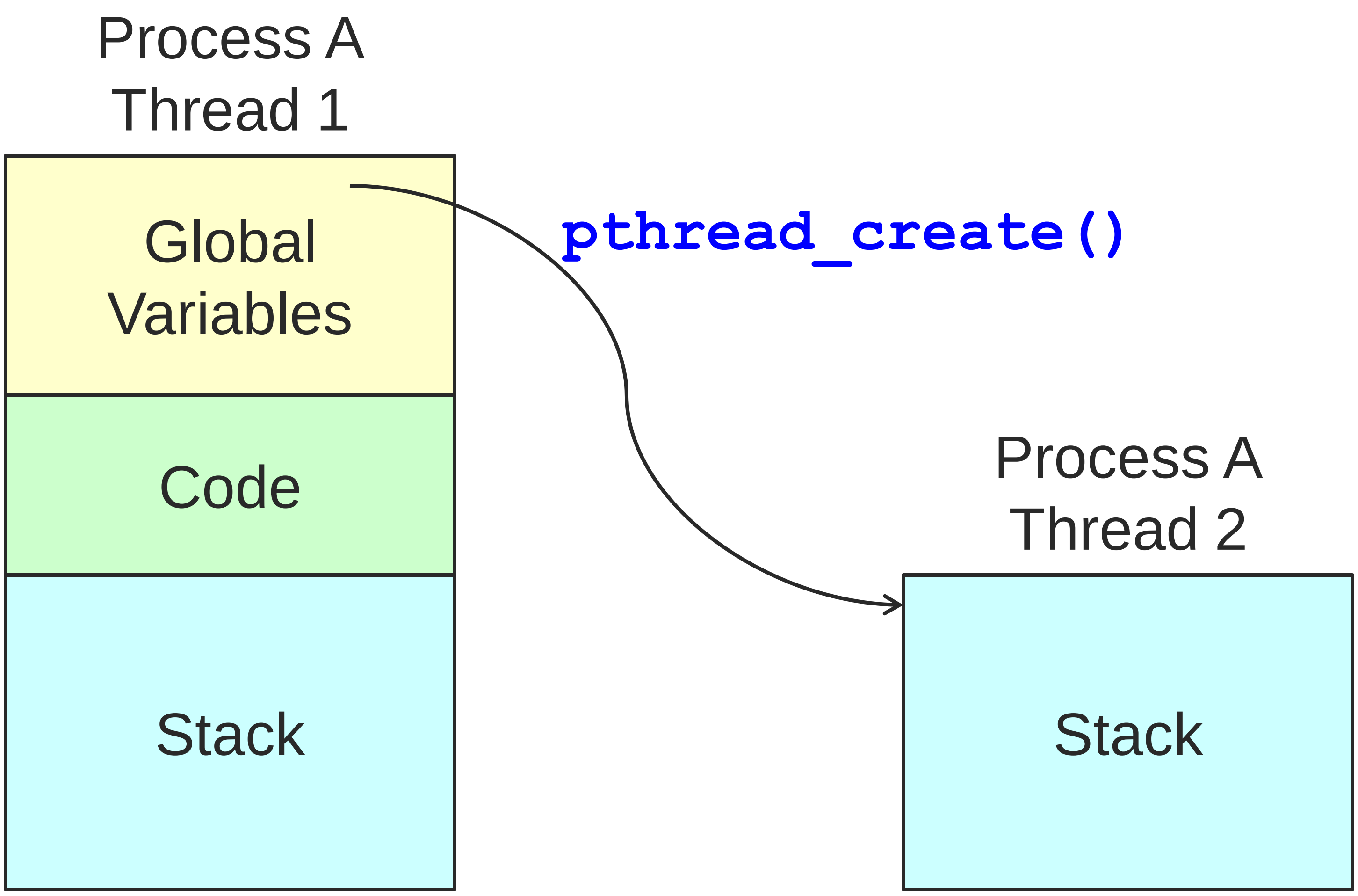
Thread vs. Process Creation

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

fork()


Process A
Global Variables
Code
Stack
fork()
Process B
Global Variables
Code
Stack

pthread_create()



Possible output?

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

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;
}
```

Possible output?

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

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

Summary: Creating Threads

- Initially, `main()` has a single thread
  - All other threads must be explicitly created
- `pthread_create()` ➔ new executable thread
  - Can be called any number of times from anywhere
- Maximum number of threads is implementation dependent
- Question:
  - After a thread has been created, how do you know when it will be scheduled to run by the operating system?
  - Answer: It is up to the operating system
    - Note: Good coding should not require knowledge of scheduling

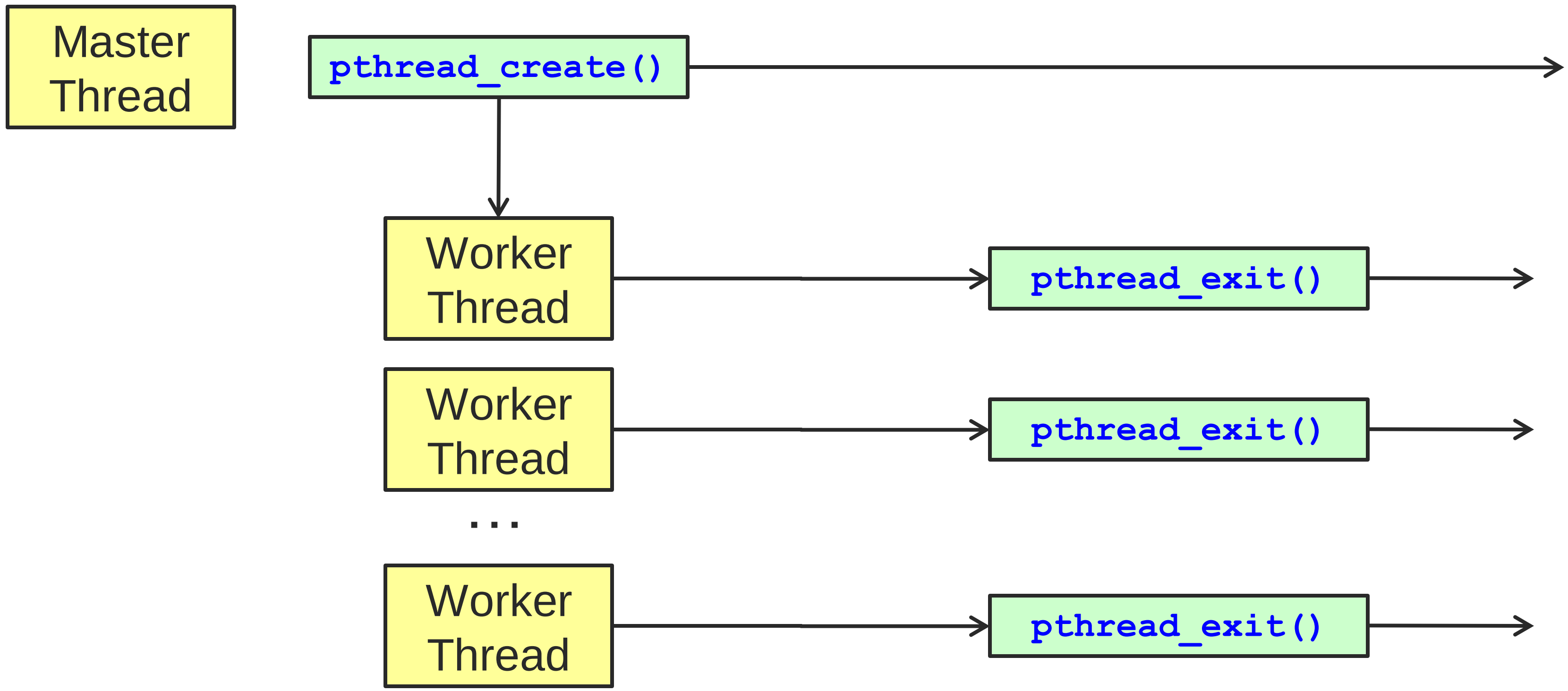
pthreads Attributes

- 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

pthreads 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`

Detached Threads

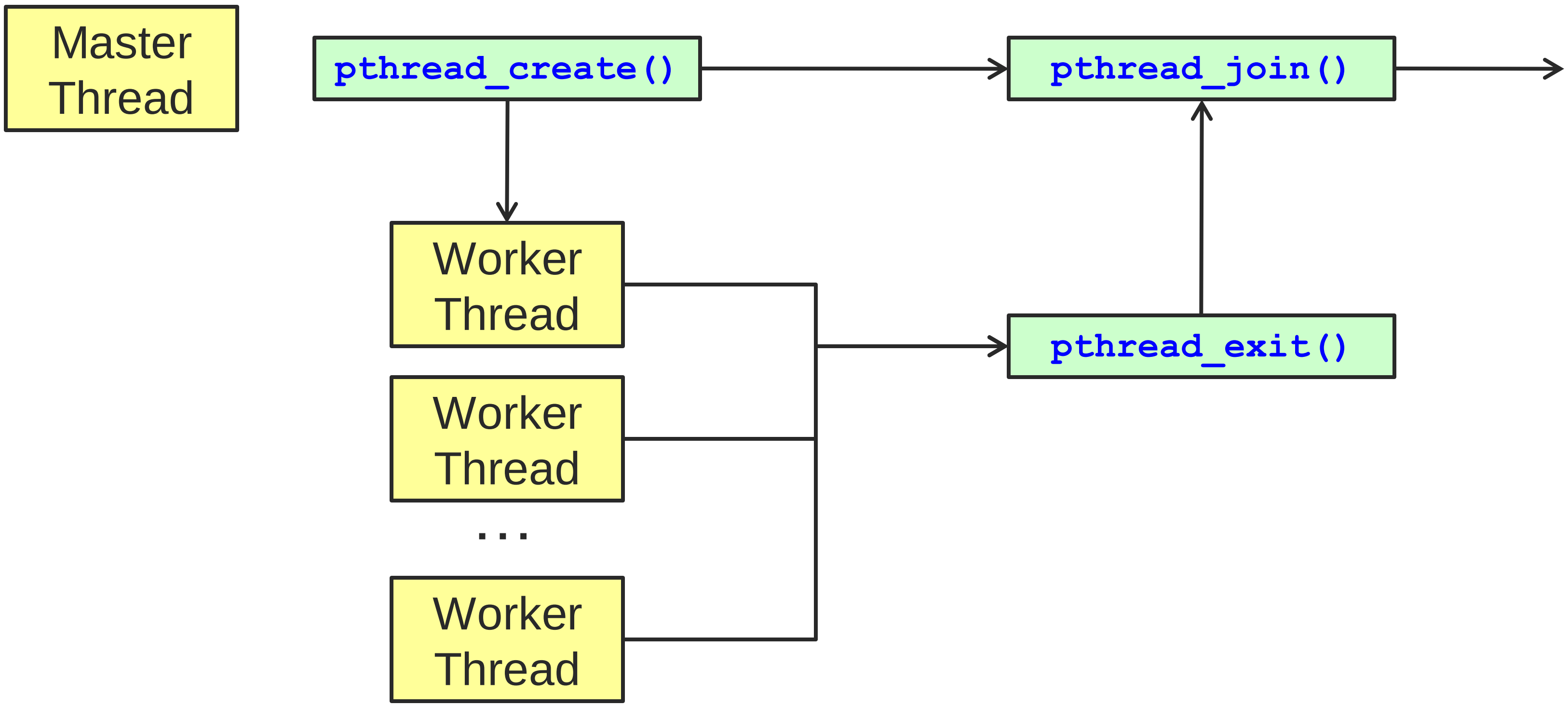


Detaching Threads: `pthread_detach()`

```
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

Joined Threads



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`**

Waiting for Threads: `pthread_join()`

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

- Note
  - You cannot call `pthread_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_init(&attr);
pthread_attr_setdetachstate(&attr,
   PTHREAD_CREATE_JOINABLE);
```

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.

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);
}
```

What is missing?

```
pthread_attr_t attr;
/* Initialize and set thread detached
   attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,
       PTHREAD_CREATE_JOINABLE);
```

What could happen without this code?

Example: pthread_join()

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 4

int main (int argc, char *argv[]) {
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc;
   long t;
   void *status;
```

```
/* Initialize and set thread detached
   attribute */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
   PTHREAD_CREATE_JOINABLE);

for(t=0; t<NUM_THREADS; t++) {
   printf("Main: creating thread %ld\n", t);
   rc = pthread_create(&thread[t], &attr,
        BusyWork, (void *)t);
   if (rc) {
      printf("ERROR; return code is %d\n",
             rc);
      exit(-1);
      }
   }
/* Free attributes */
pthread_attr_destroy(&attr);
```

Example: pthread_join()

```
void *BusyWork(void *t) {
   int i;
   long tid;
   double result = 0.0;
   tid = (long)t;
   printf("Thread %ld starting...\n",
      tid);
   for (i=0; i<1000000; i++) {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld result = %e\n",
      tid, result);
   pthread_exit((void*) t);
}
```

```
int main (int argc, char *argv[]) {
...

   /* Wait for the other threads */
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code is %d\n", rc);
         exit(-1);
      }
      printf("Main: status for thread %ld: %ld\n",
            t, (long)status);
   }

   printf("Main: program completed. Exiting.\n");
   pthread_exit(NULL);
}
```

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

Thread Lifetime

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

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

Main points

- A thread is the lightest unit of work that can be scheduled to run on the processor
- When creating a thread you
  - Indicate which function the thread should execute
  - Indicate the detach state of the thread
- When a new thread is created
  - It runs concurrently with the creating thread
  - It shares common data space

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

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

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?