ECS 150 - Concurrency and threads

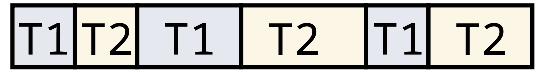
Prof. Joël Porquet-Lupine

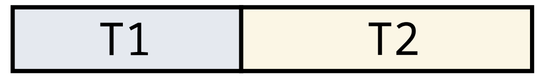
UC Davis - FQ22

Concurrency

Definition

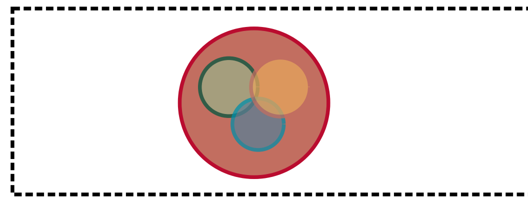
- Concurrency is the composition of independently executing tasks
    - Tasks can start, run, complete in overlapping time periods
- Opposite to *sequential execution*

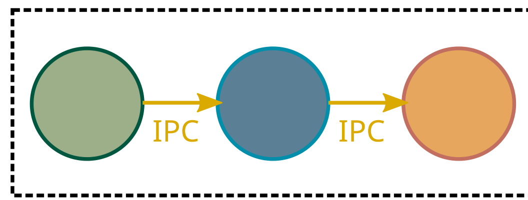




Process concurrency

- Decompose complex problems into simple(r) ones
- Make each simple one a process
- Resulting processes run *concurrently*





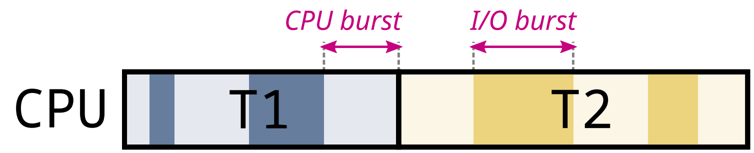
Example

- By default, gcc runs compilation tools sequentially, using intermediary files
- With option `-pipe`, gcc runs `cpp | cc1 | as | ld`
    - Tools run concurrently as independent but cooperating processes

Concurrency

Types of concurrency

- Example of sequential execution
    - CPU and I/O bursts

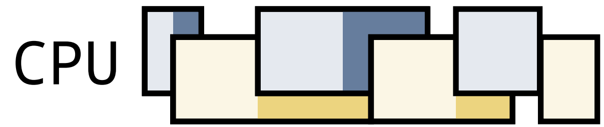


CPU virtualization

- Processes interleaved on same CPU

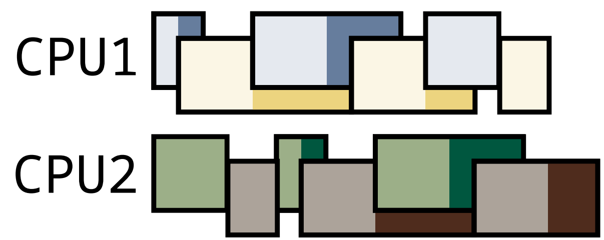
I/O concurrency

- I/O bursts overlapped with CPU bursts
- Each task runs almost as fast as if it had its own computer
- Total completion time reduced



CPU **parallelism**

- Requires multiple CPUs
- Processes running *simultaneously*
- Speedup



Concurrency

Parallelism (short digression)

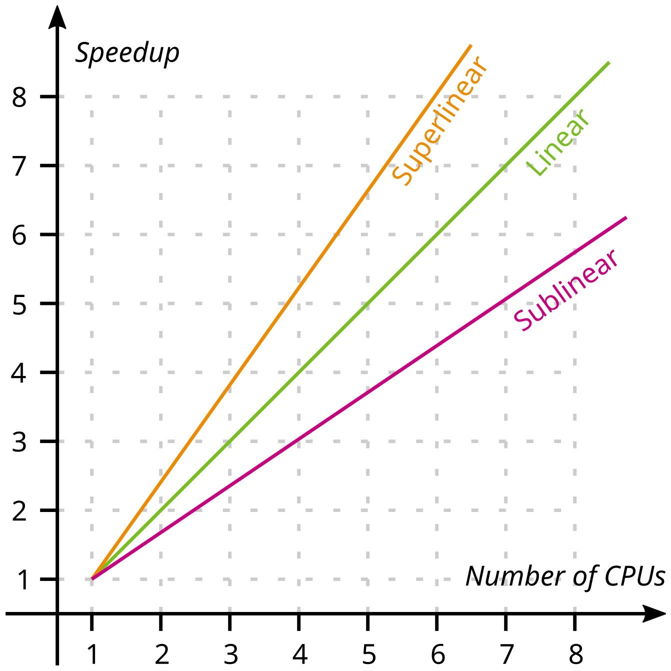
Real-life example

Parallelism is common in real life

- A single sales employee sells $1M
- Expectation that hiring another sales employee will generate $2M
  - Ideal speedup of 2

Seepdup

- Speedup can be *linear*
- More often *sublinear*
  - Bottleneck in sharing resources, coordination overhead, etc.
- Quite rarely *superlinear*
  - Caching effect, different algorithms



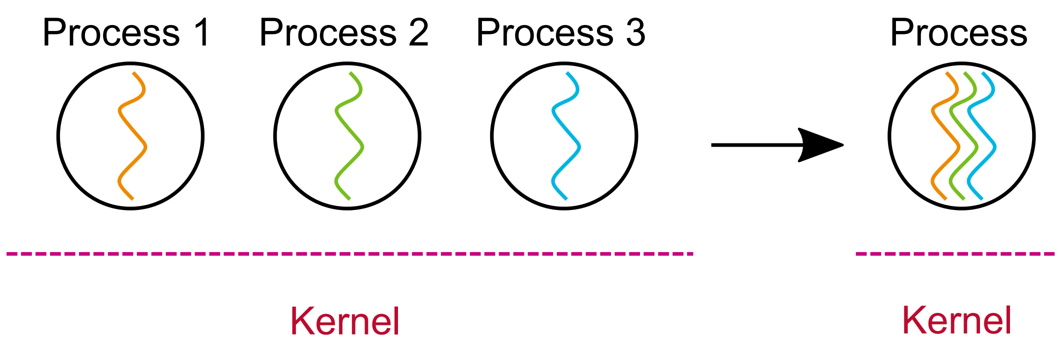
Concurrency

Process concurrency limitations

- Quite heavy for the OS to fork a process
    - Duplication of resources (address space, environment, execution flow)
- Slow context switch
    - E.g., some processor caches not shared between processes
- Difficulty to communicate between processes
    - Only IPCs, which all necessitate kernel intervention (syscalls)

Idea

- Eliminate duplication of the address space and most of the environment
- Place concurrent computations within the same address space

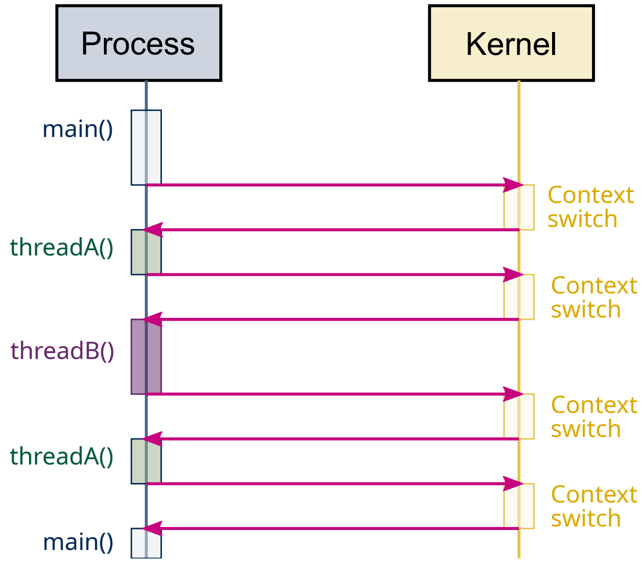


Threads: introduction

Definition

- One or more threads per process (i.e., per memory address space)
- *Single execution sequence* that represents a *separately schedulable* task
  - Familiar programming model (sequential instruction of instructions)
  - Thread can be run or suspended at any time, independently from another
- Also known as *lightweight process or task*

Example




```
int main(void) {
    statements;
    ...
    thread_create(threadA);
    thread_create(threadB);
    ...
}

void threadA(void) {
    statements;
    ...
}

void threadB(void) {
    statements;
    ...
}
```

Threads: introduction

Rationale

Problem structure

- We think linearly
- But the world is concurrent

Responsiveness

- One thread to maintain quick response with user
- Other thread(s) to execute longer tasks in the background, or block on I/O

Faster execution

- Threads scheduled across different processors in a multi-processor system
- Achieve true parallelism

Sharing and communication

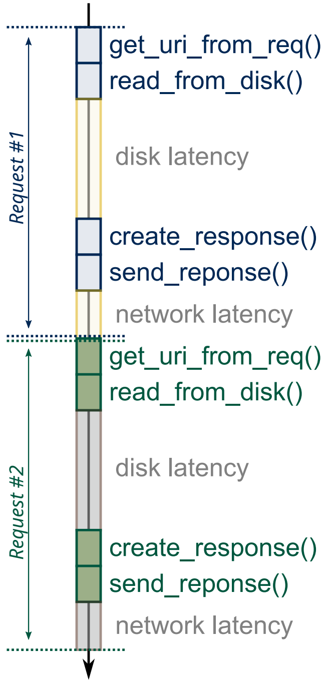
- No need for heavy IPCs
- Use of shared memory

Threads: introduction

Example 1: Web server

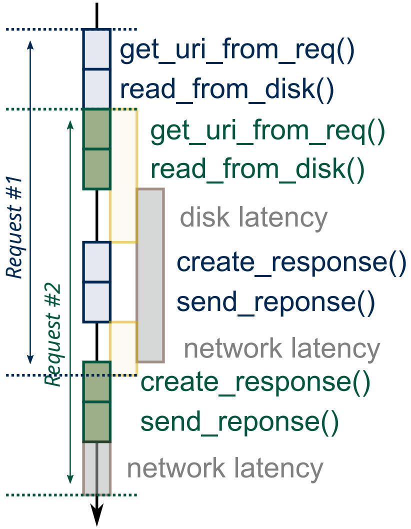
Monothreaded process

```
while (1) {
    uri = get_uri_from_req();
    data = read_from_disk(uri);
    resp = create_response(data);
    send_response(resp);
}
```



Multithreaded process

```
void webserver(void) {
        while (1) {
            uri = get_uri_from_req();
            data = read_from_disk(uri);
            resp = create_response(data);
            send_response(resp);
        }
}
int main(void) {
        for (int i = 0; i < N; i++)
            thread_create(webserver);
}
```



Threads: introduction

Example 2: Array computation

- Assuming a dual-processor system...

Monothreaded process

```
int a[n], b[n], c[n];

for (i = 0; i < n; i++)
    a[i] = b[i] * c[i];
```

CPU #1

```
a[0] = b[0]*c[0];
a[1] = b[1]*c[1];
...
a[n-1] = b[n-1]*c[n-1];
```

CPU #2

Multi-threaded process

```
void do_mult(int p, int m) {
    for (i = p; i < m; i++)
        a[i] = b[i] * c[i];
}

int main(void) {
    thread_create(do_mult, 0, n/2);
    thread_create(do_mult, n/2, n);
    ...
}
```

CPU #1

```
a[0] = b[0]*c[0];
a[1] = b[1]*c[1];
...
a[n/2-1]
    = b[n/2-1]*c[n/2-1];
```

CPU #2

```
a[n/2] = b[n/2]*c[n/2];
...
...
a[n-1] = b[n-1]*c[n-1];
```

- Parallel computation
- Not achievable by process forking

Threads: characteristics

Execution context

- Threads have the exclusive use of the processor registers while executing
- Threads each have their own stacks
  - But no memory protection from other threads of the same process
- When a thread is preempted, the registers are saved as part of its state
  - The next thread gets to use the processor registers

Process environment

- All threads of a process share the same environment
  - Current working directory
  - User ID, Group ID
  - File descriptors
  - Etc.

```c
void thread(void) {
    printf("hello\n");
    chdir("/");
}
int main(void) {
    char dir[PATH_MAX];
    ...
    dup2(STDOUT_FILENO, fd[1]);
    thread_create(thread);
    ...
    printf("%s", getcwd(dir, PATH_MAX));
    ...
}
```

Threads: characteristics

Address space

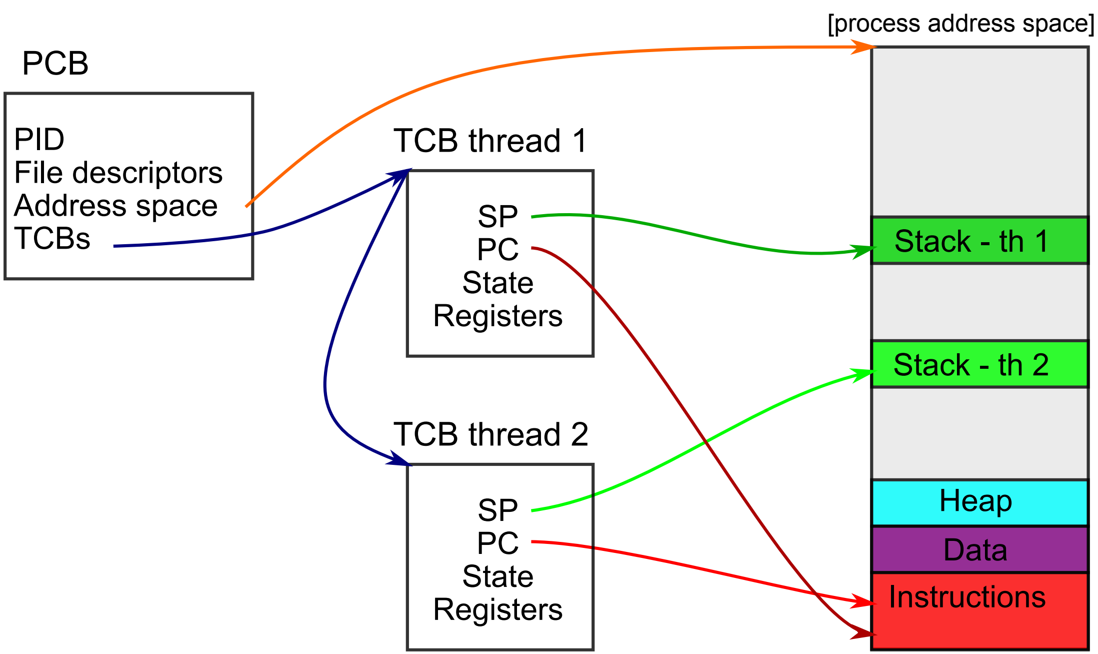
- All process data can be accessed by any thread
  - Particularly global variables
  - Heap is also be shared (via pointers)

```
int global, *a;

void thread(int arg) {
    int var = global + arg;
    *a = var;
}

int main(void) {
    global = 42;
    a = malloc(sizeof(int));
    thread_create(thread, 23);
    thread_create(thread, 42);
    ...
}
```

Threads: characteristics

Metadata structures

- *Process Control Block* (**PCB**)
  - Process-specific information
  - PID, Owner, priority, current working directory, active thread, pointers to thread control blocks, etc.
- *Thread Control Block* (**TCB**)
  - Thread-specific information
  - Stack pointer, PC, thread state, register values, pointer to PCB, etc.



Threads: characteristics

Differences between threads and processes

Thread

- Has **no** code or data segment or heap of its own. Only has its own stack and set of registers.
- Cannot live on its own: must live within a process. There can be more than one thread in a process - the original thread calls main() and retains the process's initial stack.
- If it dies, its stack is reclaimed.
- Depending on implementation, each thread can run on a different physical processor.
- Communication between threads via implicit shared address space.
- Inexpensive creation and context switch.

Process

- Has code/data/heap and other segments of its own. Also has its own registers.
- There must be at least one thread in a process. The thread that executes main() and uses the process's stack.
- If it dies, its resources are reclaimed and all its internal threads die as well.
- Each process can run on a different physical processor.
- Communication between processes through kernel-managed IPCs
- More expensive creation and context switch.

ECS 150 - Concurrency and threads

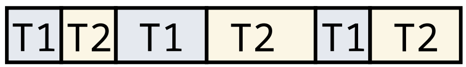
Prof. Joël Porquet-Lupine

UC Davis - FQ22

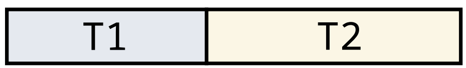
Recap

Concurrency

- Composition of independently executing tasks

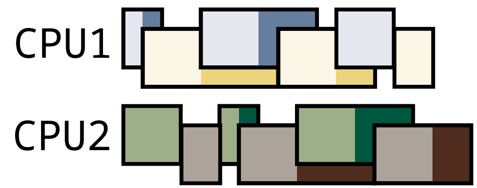


- Opposite to *sequential* execution



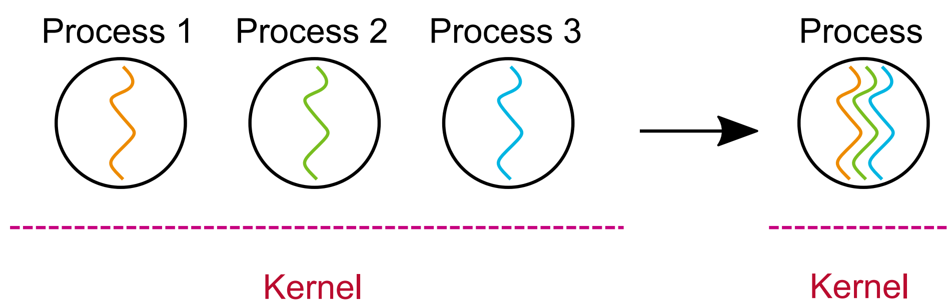
Parallelism

- Specific type of concurrency
- Requires multiple CPUs



Processes vs threads

- Process concurrency has limitations
  - Slow context switch, difficult to communicate
- Concurrent threads within same process
  - Easier communication, parallel computation



Threads: models

API

Exact API varies depending on OS/library (e.g., POSIX pthreads)

```
/* Thread function prototype */
typedef int (*func_t)(void *arg);

/* Create new thread and return its TID */
thread_t thread_create(func_t func, void *arg);

/* Wait for thread @tid and retrieve exit value */
int thread_join(thread_t tid, int *ret);

/* Yield to next available thread */
void thread_yield(void);

/* Exit and return exit value */
void thread_exit(int ret);
```

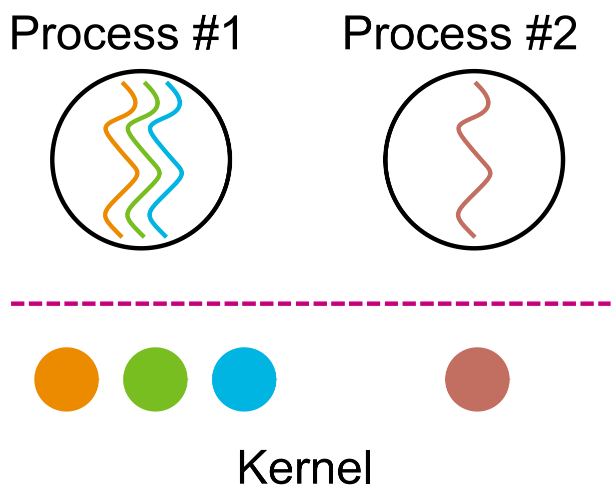
Implementation models

- Kernel-level threads (one-to-one)
- User-level threads (many-to-one)

Threads: models

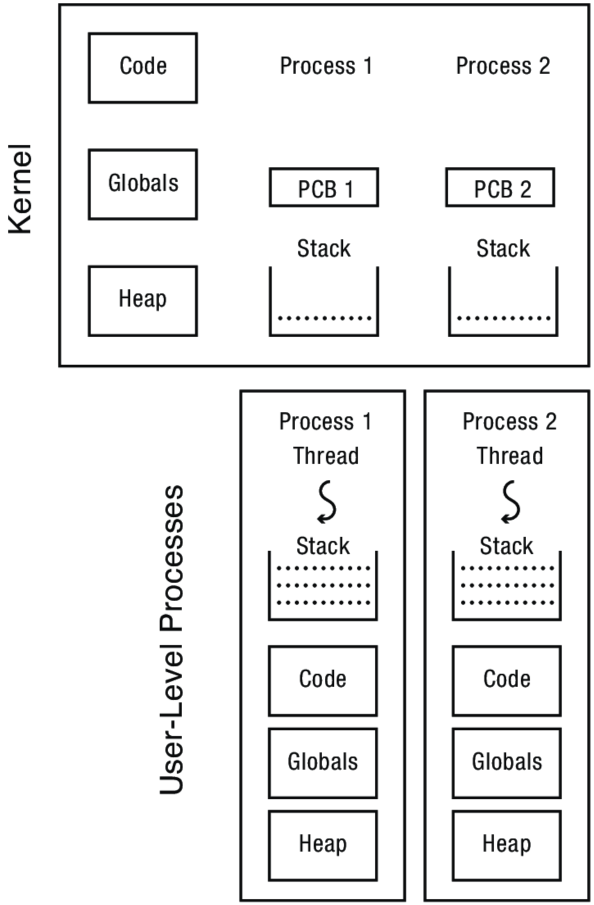
Kernel-level threads (one-to-one)

- *Kernel-level threads* are threads which the OS knows about
  - Every process is composed of a least one kernel-level thread (`main()`)
- Kernel manages and schedules threads (along with processes)
  - System calls to create, destroy, synchronize threads
  - E.g., `clone()` syscall on Linux
- Switching between threads of same process requires a light context switch
  - Values of CPU registers, PC and SP must be switched
  - Memory protection remains since threads share the same address space

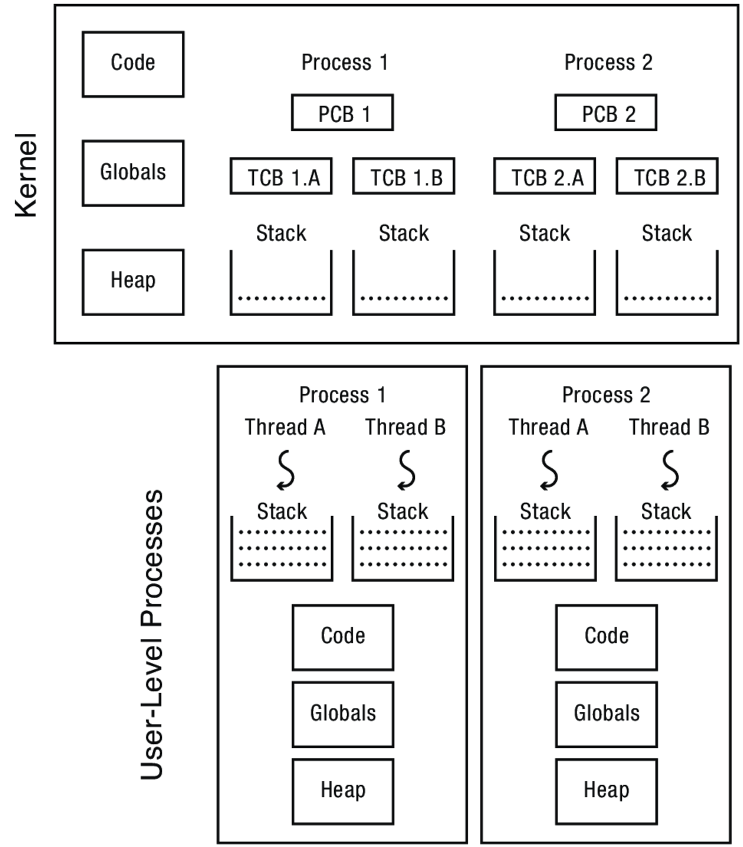


Threads: models

Single-threaded processes



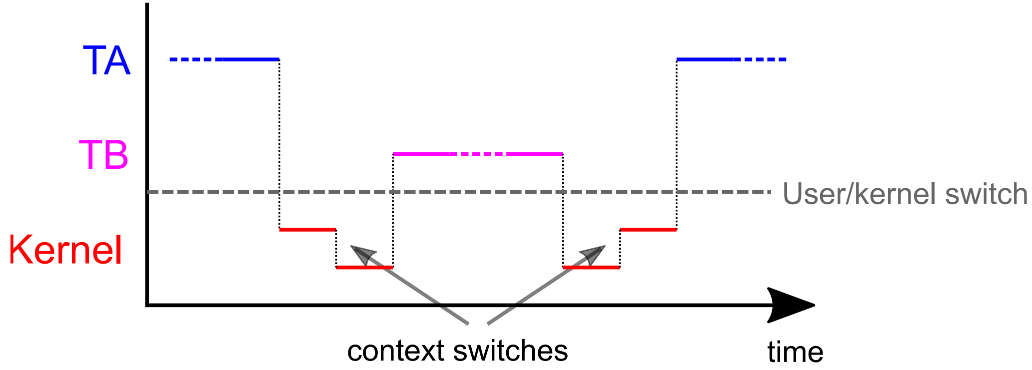
Multi-threaded processes



Threads: models

Context switch procedure

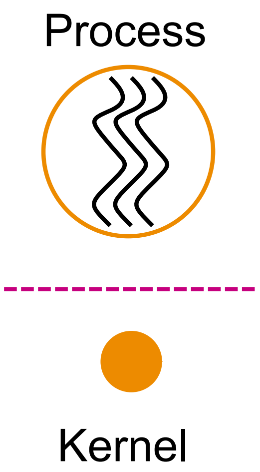
- Thread *A* stops running
  - That is, blocks (I/O), is interrupted (timer), or voluntarily yields (syscall)
  - Mode switch to kernel mode
- OS chooses new thread *B* to run
- OS switches from *A* to *B*
  - Saves thread *A*'s state (save processor registers to *A*'s TCB)
  - Restore new thread *B*'s state (restore processor registers from *B*'s TCB)
- OS returns to thread *B*
  - Mode switch to user mode
- New thread *B* is running



Threads: models

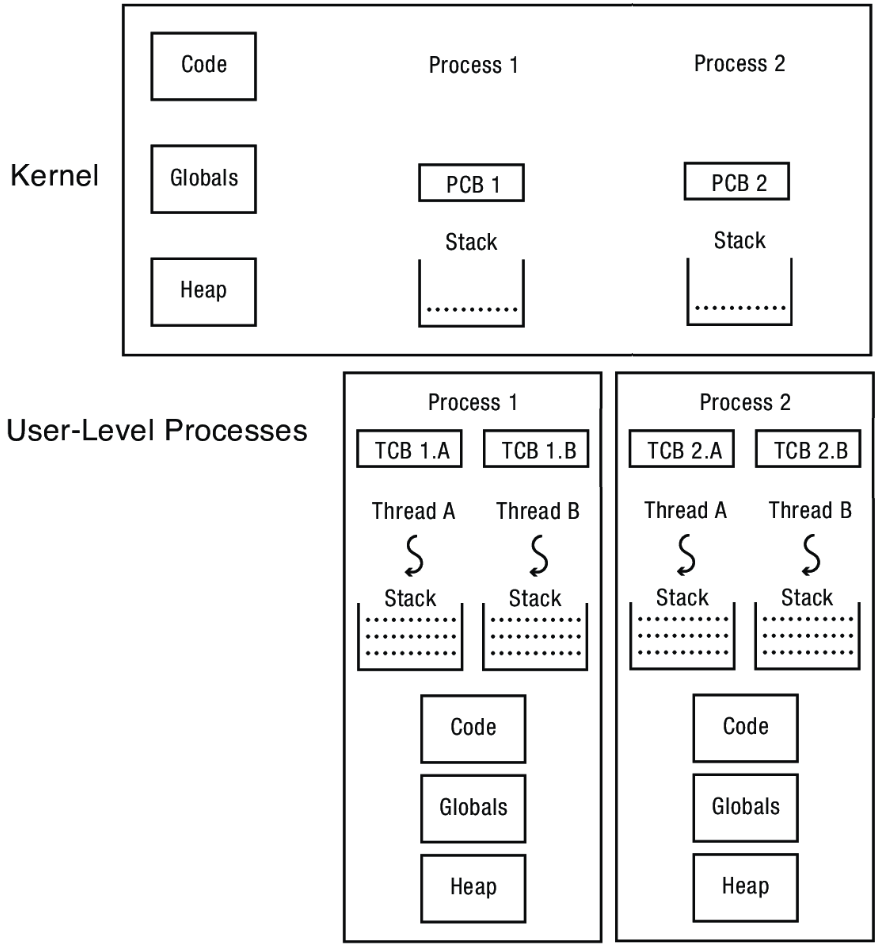
User-level threads (many-to-one)

- *User-level threads* are threads which the OS does **not** know about
  - OS only knows and schedules processes, not threads within processes
- Programmer uses a dedicated *thread library* to manage threads
  - Functions to create, destroy, synchronize threads
  - User-level code can define scheduling policy
- Switching between threads doesn't involve a (kernel-managed) context switch



Threads: models

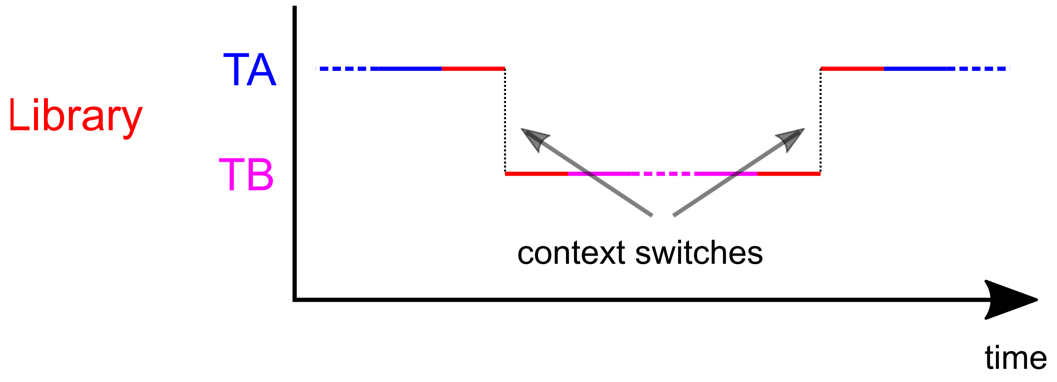
Multi-threaded processes at user-level



Threads: models

Context switch procedure (sort of)

- Thread *A* is running
- Thread *A* voluntarily yields (function call), or is disrupted (by signal)
- Library picks new thread *B* to run
- Library saves thread *A*'s state (to *A*'s custom TCB)
- Library restores thread *B*'s state (from *B*'s custom TCB)
- New thread *B* is running



Pitfall

- Whole process is blocked if one thread blocks on I/O

Threads: models

Differences between kernel- vs user-level threads

Kernel-level thread

Pros

- Blocking system calls suspend the calling thread only (I/O)
- Threads can run simultaneously on a multiprocessor system
- Signals can usually be delivered to specific threads
- Used by existing systems, e.g. Linux

Cons

- Can be heavy, not as flexible

User-level thread

Pros

- Really fast to create and switch between threads (no system calls or full context switches necessary)
    - May be an order of magnitude faster
- Customizable scheduler

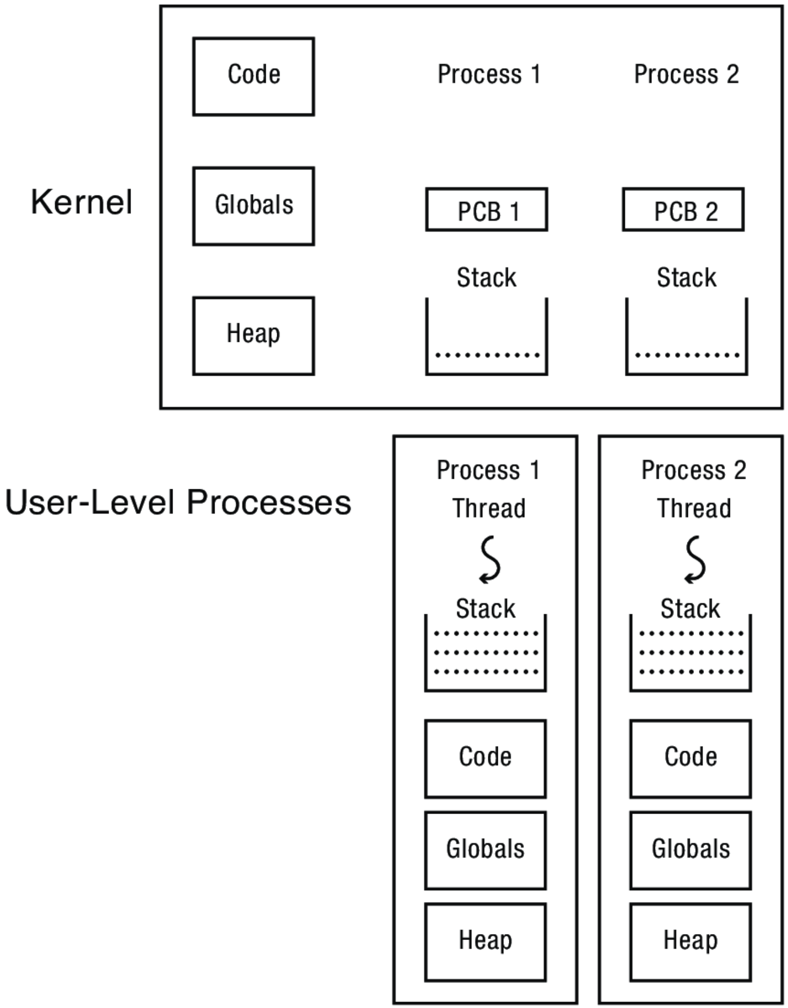
Cons

- All threads of process are blocked on system calls
    - Can use non-blocking versions, if they exist
- Customizable scheduler

[Why you can have millions of Goroutines but only thousands of Java Threads]

Threads: models

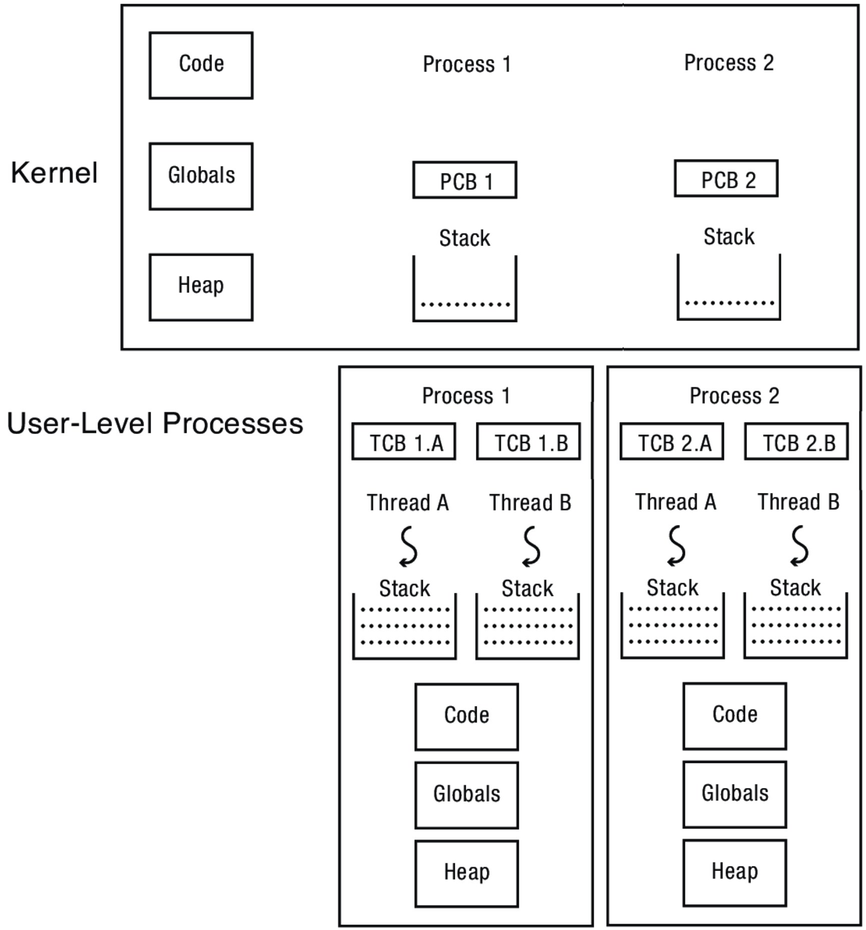
One abstraction, many flavors



1. Single-threaded processes
   - Traditional application model
   - Mapping 1:1 with kernel-level threads
2. Multi-threaded processes with user-level threads
3. Multi-threaded processes with kernel-level threads
4. In-kernel threads (aka *kernel threads*)

Threads: models

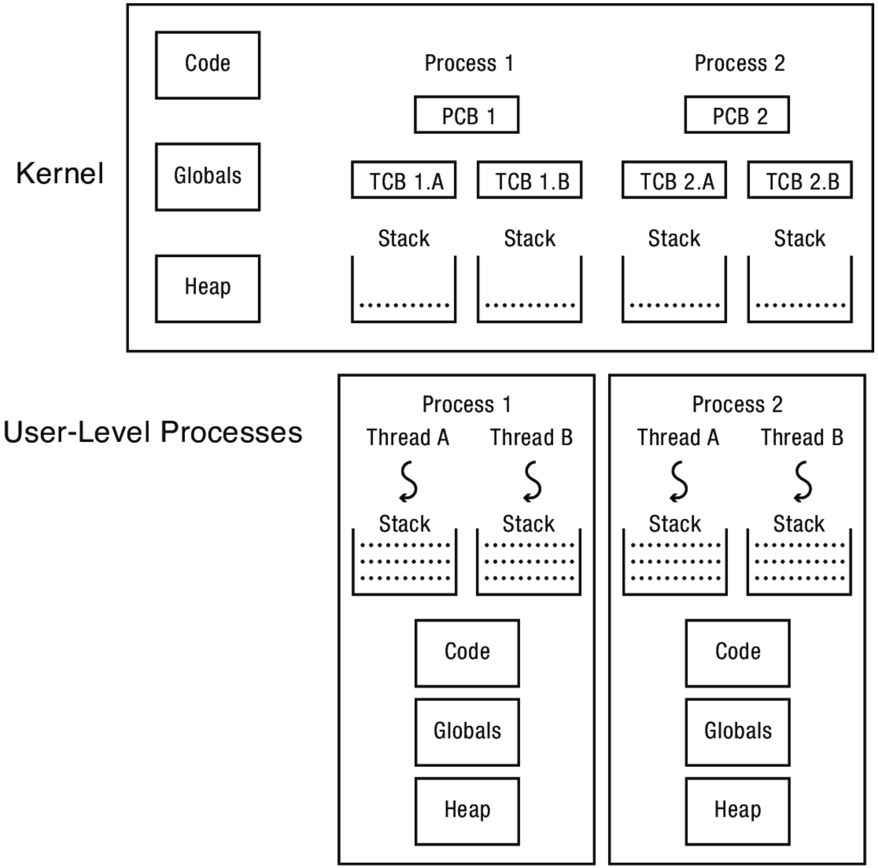
One abstraction, many flavors



1. Single-threaded processes
2. Multi-threaded processes with user-level threads
   - Threads are managed in user-space
   - Mapping M:1 with kernel-level threads
   - Thread management through function calls
   - Scheduled by user-space library scheduler
   - TCBs in user-space library data structures
3. Multi-threaded processes with kernel-level threads
4. In-kernel threads (aka *kernel threads*)

Threads: models

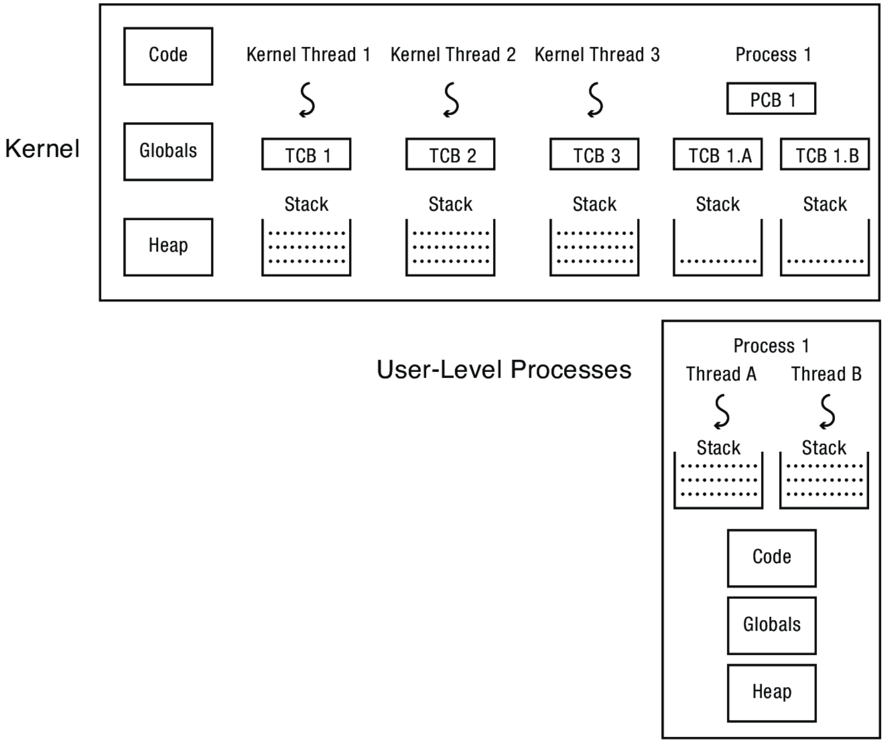
One abstraction, many flavors



1. Single-threaded processes
2. Multi-threaded processes with user-level threads
3. Multi-threaded processes with kernel-level threads
   - Threads are managed by OS
   - Thread management through system calls
   - TCBs and PCBs in kernel
   - Scheduled by kernel scheduler
4. In-kernel threads (aka *kernel threads*)

Threads: models

One abstraction, many flavors



1. Single-threaded processes
2. Multi-threaded processes with user-level threads
3. Multi-threaded processes with kernel-level threads
4. In-kernel threads (aka *kernel threads*)
   - The kernel itself can be multi-threaded
   - E.g., idle thread, thread migration, OOM reaper, disk writeback, etc.

Threads: models

One more flavor: cooperative vs preemptive

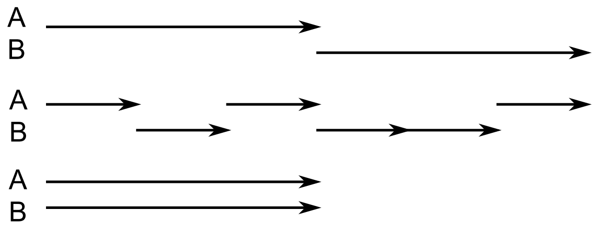
Cooperative

- Threads run until they yield control to another thread (aka *fibers*)
  - The action of yielding is in the code itself
- Better control of scheduling
- Simpler reasoning about shared resources
- Can lead to potential starvation on mono-core machines
- Need to reason further for multi-core machines

Preemptive

- Threads are frequently interrupted and forced to yield
- Certain guarantee of fairness
- Scheduling is not deterministic
- Resource sharing needs to be resistant to preemption

Indeterministic scheduling



Threads: models

POSIX threads

- POSIX 1003.1c (aka pthreads) is an API for multithreaded programming standardized by IEEE as part of the POSIX standards
- Multithreaded programs using pthreads are likely to run unchanged on a wide variety of UNIX-based systems

Interface

- Only defines the interface
  - user-space implementation
  - or kernel space implementation

```c
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
void pthread_exit(void *status);
int pthread_join(pthread_t thread, void **status);
pthread_t pthread_self(void);
int pthread_equal(pthread_t thread_1, pthread_t thread_2);
...
```

Threads: pitfalls/issues

Forking

```c
void *thread_fcn(void *ptr)
{
    char *msg_thread = (char*)ptr;
    fork();
    printf("%s\n", msg_thread);
}

int main(void)
{
    pthread_t t1;
    char *msg_main = "Hello";

    pthread_create(&t1, NULL, thread_fcn,
                   (void*)msg_main);

    pthread_join(t1, NULL);

    printf("Done!\n");

    return 0;
}
```

```
$ ./a.out
Hello
Hello
Done!
```

- fork() only clones the calling thread
- Mixing multi-threading and forking is not recommended as it can lead to undesirable situations

Threads: pitfalls/issues

Sharing

- Shared data structures and files

```c
int a;

void *thread_fcn(void *ptr)
{
    a++;
}

int main(void)
{
    pthread_t t1, t2;

    pthread_create(&t1, NULL,
                   thread_fcn, NULL);
    pthread_create(&t2, NULL,
                   thread_fcn, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("a=%d\n", a);

    return 0;
}
```

- May lead to surprising results...

```
$ ./a.out
a = 2
$ ./a.out
a = 1
```

- Will require use of synchronization mechanisms
- Will see that in next topic!

Threads: pitfalls/issues

Signals

```
void sigsegv_hdl(int sig, siginfo_t *siginfo,
                 void *context)
{
    ucontext_t *c = (ucontext_t*)context;
    ...
}

void *thread_fcn(void *ptr)
{
    *(NULL) = 42;
}

int main(void)
{
    struct sigaction act = { 0 };
    pthread_t t1;

    /* Install handler for segfaults */
    act.sa_sigaction = &sigsegv_hdl;
    act.sa_flags = SA_SIGINFO;
    sigaction(SIGSEGV, &act, NULL);

    pthread_create(&t1, NULL,
                   thread_fcn, NULL);
    pthread_join(t1, NULL);

    return 0;
}
```

- Signals can target specific threads (e.g., SIGSEGV)
- Or target the entire process
    - Sent to first thread that doesn't block them (e.g., SIGINT)

Threads: pitfalls/issues

Libraries

- Global variables and non-reentrant library functions

```c
void *thread_fcn(void *ptr)
{
    char *msg = (char*)ptr;
    char *p = strtok(msg, " ");
    while (p) {
        if (write(1, p, strlen(p)) == -1)
            perror("write");
        p = strtok(NULL, " ");
    }
}

int main(void)
{
     pthread_t t1, t2;
     char msg1[] = "Thread #1";
     char msg2[] = "Thread #2";

     pthread_create(&t1, NULL, thread_fcn,
                    (void*)msg1);
     pthread_create(&t2, NULL, thread_fcn,
                    (void*)msg2);

     ...
}
```

- Functions should be *reentrant*
  - e.g., `strtok_r()`
  - No shared context between threads
- Global variables: *Thread Local Storage*
  - C11 extension
  - `__thread int errno;`
  - Provides one global variable copy per thread