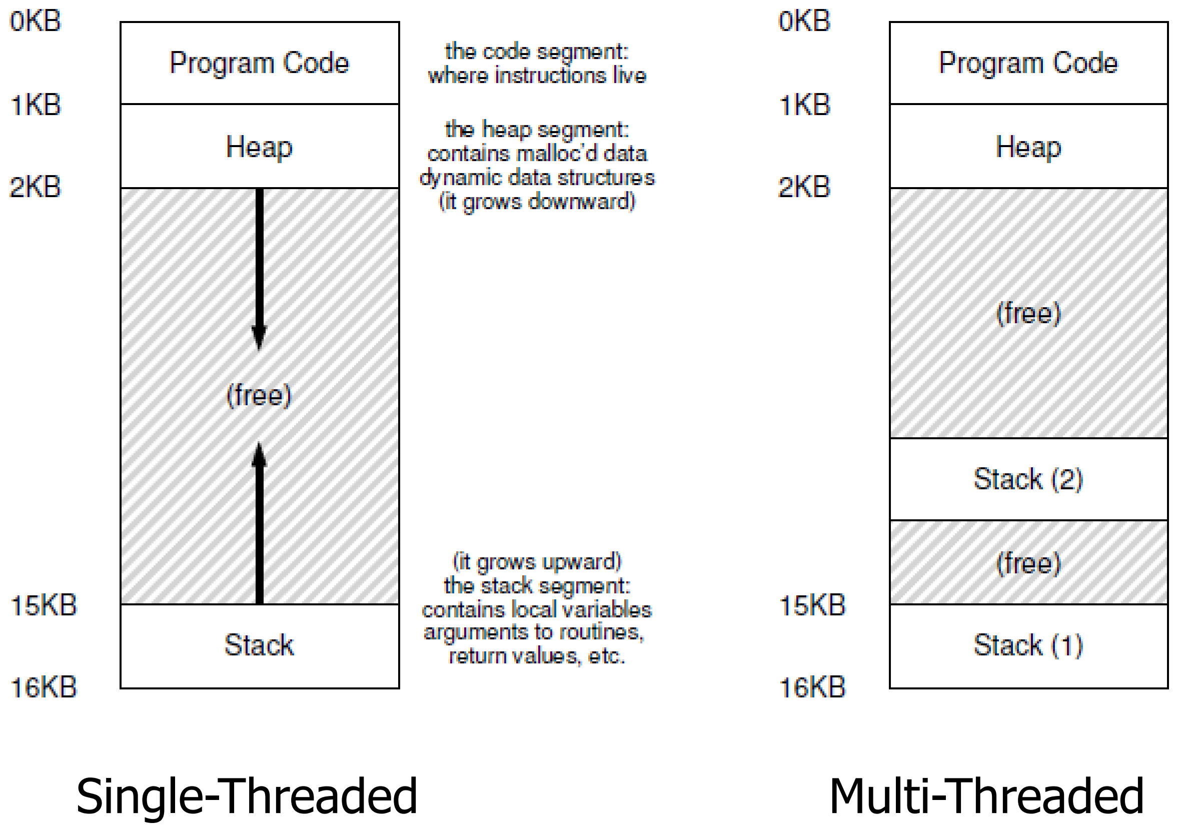
Concurrency, Thread

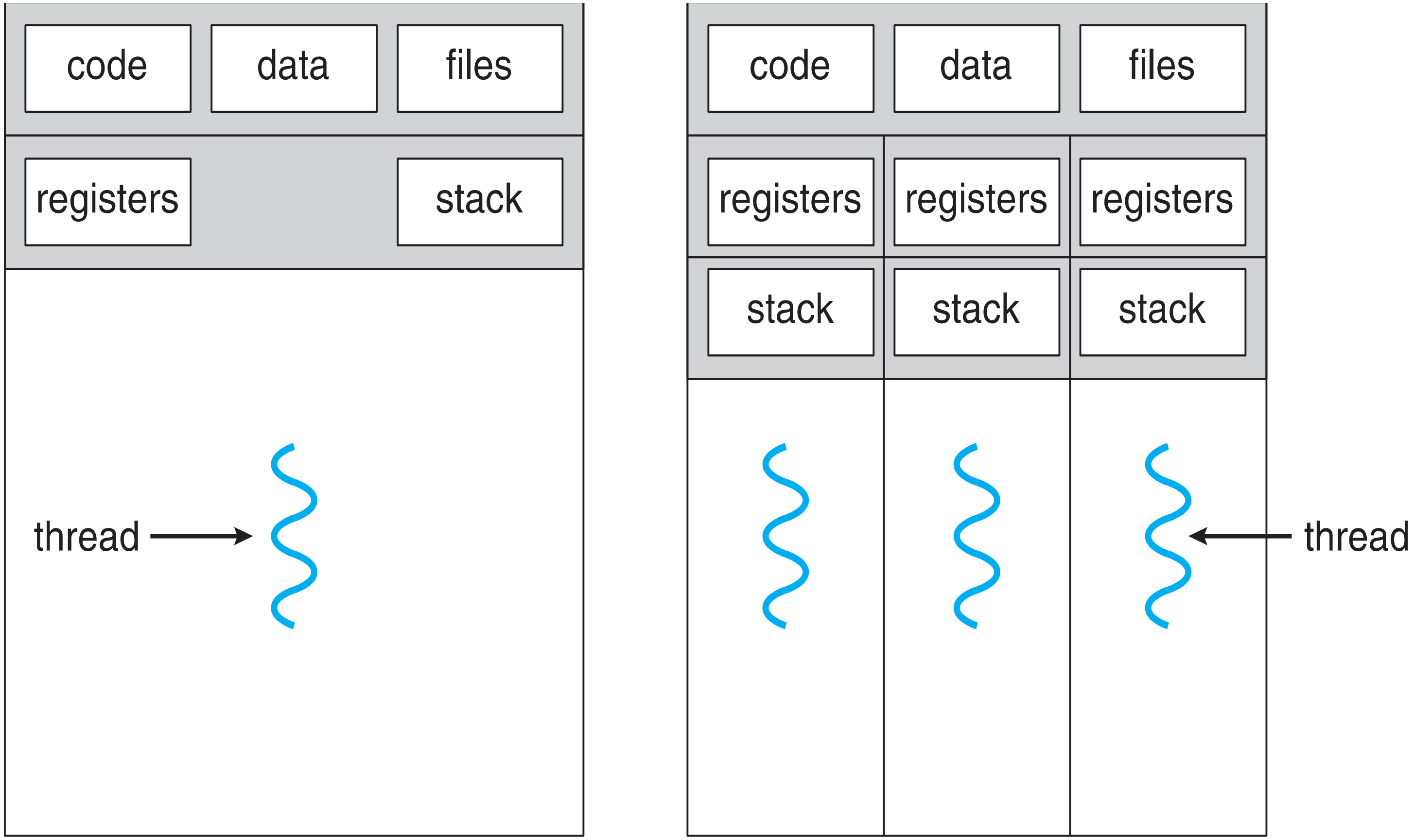
Thread

- Classic view
  - a single point of execution within a program
  - a single PC where instructions are being fetched from and executed),
- Multi-threaded program
  - Has more than one point of execution
  - multiple PCs, each of which is being fetched and executed from.
- Threads share the same address space and thus can access the same data
- Each thread has its own private set of registers (including PC)
- When switching from running one (T1) to running the other (T2), a **context switch** must take place
  - thread control blocks (TCBs)
  - the address space remains the same (i.e., no need to switch the page table).

Multiple Stacks



Single and Multithreaded Processes



single-threaded process

multithreaded process

Why Use Threads?

- **Parallelism**
  - One thread per CPU can make programs run faster on multiple processors
- **I/O overlapping (even in single processor)**
  - Avoid blocking program progress due to slow I/O
  - While one thread in your program waits (i.e., blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful.
  - Similar to the effect of multiprogramming

- You could use multiple processes instead of threads.
  - Threads share an address space and thus make it easy to share data
  - Processes are a more sound choice for logically separate tasks

Thread

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Benefits

- **Responsiveness**
  - may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing**
  - threads share resources of process, easier than shared memory or message passing
- **Economy**
  - cheaper than process creation, thread switching lower overhead than context switching
- **Scalability**
  - process can take advantage of multiprocessor architectures

Thread Creation

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Once a thread is created, it may start running right away, or ready

pthread join() waits for a particular thread to complete

Indeterminate executions

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Why It Gets Worse: Shared Data

```c
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

[expected]

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

even indeterminate

[real]

Uncontrolled Scheduling

counter = counter + 1;

```
100     mov 0x8049a1c, %eax
105     add  $0x1, %eax
108     mov %eax, 0x8049a1c
```

OS	Thread 1	Thread 2	(after instruction) PC	%eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

each thread has its own private registers

Race Condition

- **Race condition**
  - Results depend on the timing execution of the code, indeterminate
- **Critical section**
  - a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.
- **Mutual exclusion**.
  - guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

Edsger Dijkstra
Turing award 1972

Wish For Atomicity

- We need more powerful instructions
  - do exactly whatever we needed done in a single step ➔ atomic
  - remove the possibility of an untimely interrupt
  - E.g. memory-add 0x8049a1c, $0x1

- A few **synchronization primitives**.

One More Problem: Waiting For Another

- Another common interaction
  - One thread must wait for another to complete some action before it continues
  - e.g., when a process performs a disk I/O and is put to sleep; when the I/O completes, the process needs to be roused from its slumber so it can continue.

- **Condition variables**

Thread Libraries

- **Thread library provides programmer with API for creating and managing threads**
- **Two primary ways of implementing**
  - Library entirely in user space
    - Function call, not system call
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- *Specification*, not *implementation*
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- Compile
  - prompt> gcc -o main main.c -Wall -pthread

Thread API

- Thread Creation
- Thread Completion
- Locks
- Condition Variables

Thread Creation

```
#include <pthread.h>
int
pthread_create(      pthread_t *         thread,
               const pthread_attr_t *    attr,
                     void *              (*start_routine)(void*),
                     void *              arg);
```

- Arguments
  - thread: a pointer to a structure of type pthread_t
    - Used to interact with this thread
  - attr: specify any attributes this thread might have
    - e.g., stack size or scheduling priority
    - initialized with a separate call to pthread_attr_init()
  - function pointer: which function should this thread start running in?
  - arg: the argument to be passed to the function where the thread begins execution
    - void pointer allows us to pass in any type of argument

Thread Creation

```c
#include <pthread.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p;
    int rc;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

unpack the arguments as desired

package into a single type

Thread Completion

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

- Wait for a thread to complete
- Arguments
  - pthread_t: specify which thread to wait for.
  - a pointer to the return value

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = Malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

int
main(int argc, char *argv[]) {
    pthread_t p;
    myret_t *m;

    myarg_t args = {10, 20};
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &m);
    printf("returned %d %d\n", m->x, m->y);
    free(m);
    return 0;
}
```

Thread Completion

- don’t return a pointer which refers to something allocated on the thread’s call stack
  - automatically deallocated

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

Locks

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Providing mutual exclusion to a critical section via **locks**

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

- If no other thread holds the lock, the thread will acquire the lock and enter the critical section.
- If another thread does indeed hold the lock, the thread trying to grab the lock will not return from the call until it has acquired the lock
- Lock initialization
  - Static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
  - Dynamic int rc = pthread_mutex_init(&lock, NULL);
    assert(rc == 0); // always check success!
- Lock destroy
  - pthread_mutex_destroy()

trylock

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abs_timeout);
```

- Returns failure if the lock is already held
- the timedlock version of acquiring a lock returns after a timeout or after acquiring the lock, whichever happens first.
- avoid getting stuck (perhaps indefinitely) in a lock acquisition routine (prevent deadlock).

Condition Variables

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

- To use a condition variable, one has to in addition have a lock that is associated with this condition
  - Prevent a race condition
- Wait
  - puts the calling thread to sleep, and thus waits for some other thread to signal it, usually when something in the program has changed that the now-sleeping thread might care about.
  - Release the lock before sleep, re-acquires the lock before returning
  - For safety, the waiting thread has to re-check the condition after wake-up (There could be a spurious wake-up)

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

Spin

```
while (ready == 0); // spin
```

instead of wait()

```
ready = 1;
```

instead of signal()

- A simple flag instead of a condition variable and associated lock
- Don't ever do this
  - Poor performance
  - Error prone

User Threads and Kernel Threads

- **User threads**
  - management done by user-level threads library
  - Three primary thread libraries:
    - POSIX **Pthreads**
    - Windows threads
    - Java threads

- **Kernel threads**
  - Supported by the Kernel
  - Examples – virtually all general purpose operating systems, including: Windows, Solaris, Linux, Tru64 UNIX, Mac OS X

Implicit Threading

- **Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads**
- **Creation and management of threads done by compilers and run-time libraries rather than programmers**
- **Three methods explored**
  - Thread Pools
  - OpenMP
- **Other methods**
  - Microsoft Threading Building Blocks (TBB)
  - `java.util.concurrent` package

Thread Pools

- **Create a number of threads in a pool where they await work**
- **Advantages:**
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e. Tasks could be scheduled to run periodically
- **Windows API supports thread pools:**

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
     * this function runs as a separate thread.
     */
}
```

OpenMP

- **Set of compiler directives and an API for C, C++, FORTRAN**
- **Provides support for parallel programming in shared-memory environments**
- **Identifies parallel regions – blocks of code that can run in parallel**

```
#pragma omp parallel
```

Create as many threads as there are cores

```
#pragma omp parallel for
for(i=0;i<N;i++) {
     c[i] = a[i] + b[i];
}
```

Run for loop in parallel

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

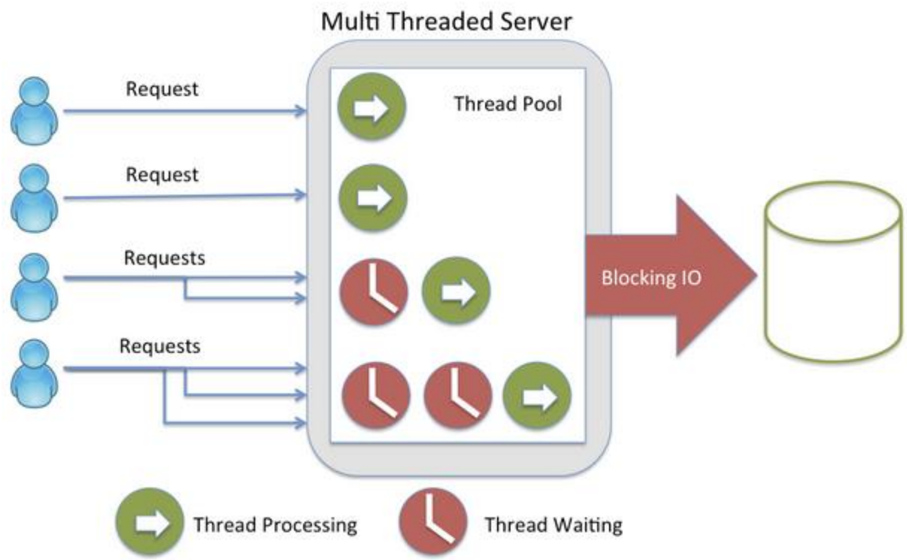
Linux Threads

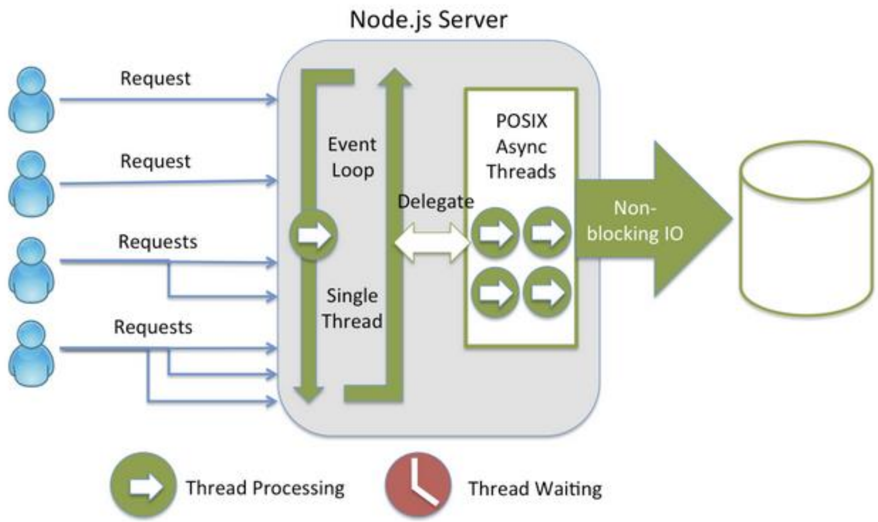
- **Linux refers to them as *tasks* rather than *threads***
- **Thread creation is done through `clone()` system call**
- `clone()` **allows a child task to share the address space of the parent task (process)**
  - for thread creation (cf> fork for process creation)
  - Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- `struct task_struct` **points to process data structures (shared or unique)**

Multithreaded Server Architecture





Homework

- Homework in Chap 27 (Debugging Race/Deadlock w/ *helgrind*)