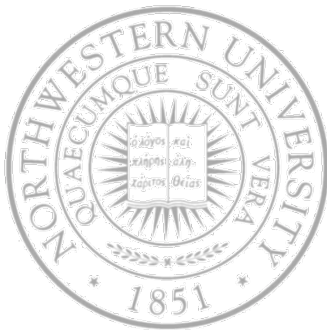


Threads



Today

- Why threads?
- Thread model & implementation
- ...

Next time

- CPU Scheduling

What's in a process

- A process consists of (at least):
 - An address space
 - Code and data for the running program
 - Thread state
 - An execution stack and stack pointer (SP)
 - Traces state of procedure calls made
 - The program counter (PC), indicating the next instruction
 - A set of general-purpose processor registers and their values
 - A set of OS resources
 - open files, network connections, ...
- A lot of concepts bundled together!

Cooperating, *concurrent* tasks

- Many programs need to perform mostly independent tasks that do not need to be serialized, e.g.
 - Web server – clients' requests, cart update, credit card checks, ...
 - Text editor – update screen, save file, spell check, ...
 - Web client – multiple request for each piece of a site
 - Parallel program – large matrix multiplication in blocks
 - ...
- Concurrency and parallelism
 - Concurrency – what's possible with infinite processors; for convenience
 - Parallelism – Your actual degree of parallel execution; for performance

How can we get this?

- Given the process abstraction as we know it
 - Fork several processes
 - Make each to map to the *same* address space to share data
 - See the `shmget()` system call for one way to do this (kind of)
- Not very efficient
 - Space: PCB, page tables, etc.
 - Time: Creating OS structures, fork and copy addr space, etc.
- Other equally bad alternatives for some of the cases
 - Entirely separate web servers
 - Finite-state machine or event-driven – a single process and asynchronous programming (non-blocking I/O)

What is actually needed

- In each examples
 - Everybody wants to run the same code
 - ... wants to access the same data
 - ... has the same privileges
 - ... uses the same resources (open files, net connections, etc.)
- But ... *wants its own HW execution state*
 - An execution stack & SP
 - PC indicating the next instruction
 - A set of general-purpose processor registers & their values

The thread model

- Traditionally
 - Process = resource grouping + execution stream
 - Resources: program text, data, open files, child processes, pending alarms, accounting info, ...
- Key idea with threads
 - Separate the concept of a process (address space, etc.)
 - From that of a minimal “thread of control” (execution state)
 - Threads are concurrent executions sharing an address space (and some OS resources)

Threads and processes

- Most modern OS's support both entities
 - Process – defines address space and global process attributes
 - Thread – a sequential execution stream *within a process*
- A thread is bound to a process/address space
 - Address space provides isolation
 - *If you can't name it, you can't use it* (read or write)
 - So, communication between processes is difficult (you have to involve the OS), but sharing data between threads is cheap
- Threads become the unit of scheduling
 - Process / address spaces are just containers where threads execute

If you haven't seen one

```
#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("begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);

    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("end\n");
    return 0;
}
```

Main	thread	thread
"begin"		
Create T1		
Create T2		
	"A"	
		"B"
Wait on T1		
Wait on T2		
"end"		

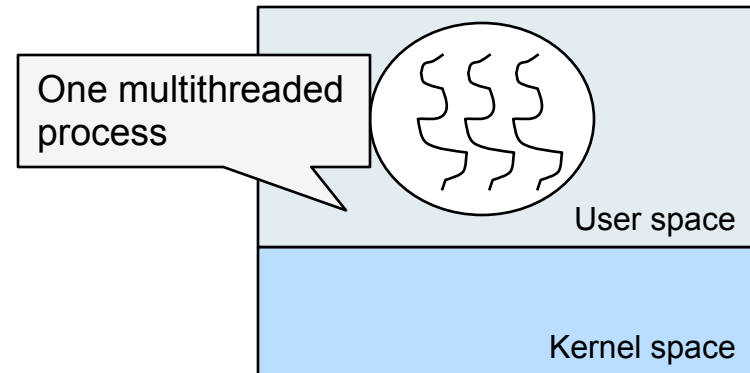
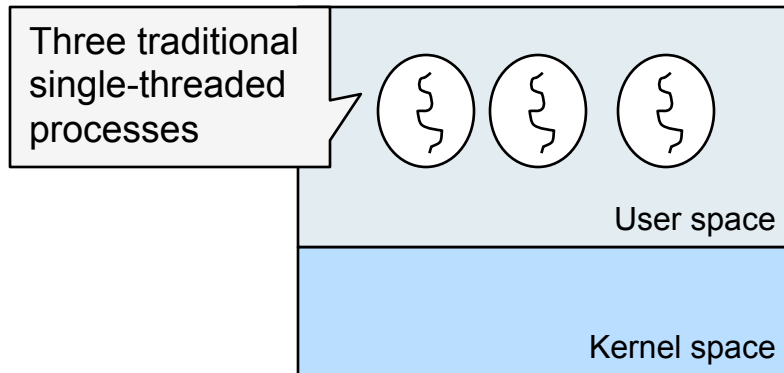
```
% gcc -o createThread createThread.c -pthread
```


Benefits of threads

- Simpler programming model for concurrent activities
 - Multiple asynchronous events, handle with separate threads using a synchronous programming model
- Easier/faster to communicate between threads than processes
- Easier/cheaper to create/destroy than processes since they have no resources attached to them
- With good mix of CPU and I/O bound activities, better performance
- Even better if you have multiple CPUs

The classical thread model

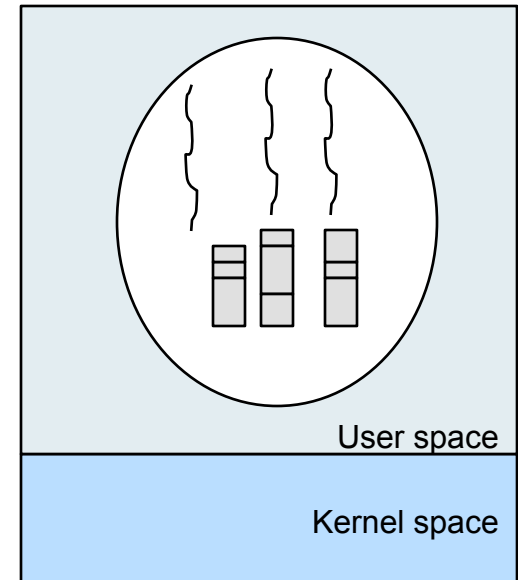
- Threads and processes



- Threads states ~ processes states
- Threads are not as independent as processes
 - All share same address space so they all can read, write or delete each other's stacks
 - There's no protection between threads (*Should they be?*)
 - Also share set of open files, child processes, signals, etc
 - If one thread opens a file, the file is visible to the others

The classical thread model

- Remember the per thread items
 - Program counter, registers, *stack*, state
 - Each thread's stack contains one frame for each procedure called but not yet returned from
- Typical thread calls



Thread call	Description
pthread_create	Create a new thread
pthread_exit	Terminate the calling thread
pthread_join	Wait for a specific thread to exit
pthread_yield	Release the CPU to let another thread run

Another simple example

```
int r1 = 0, r2 = 0;

void do_one_thing(int *ptimes)
{
    int i, j, x;

    for (i = 0; i < 4; i++) {
        printf("doing one\n");
        for (j = 0; j < 1000; j++)
            x = x + i;
        (*ptimes)++;
    } /* do_one_thing! */

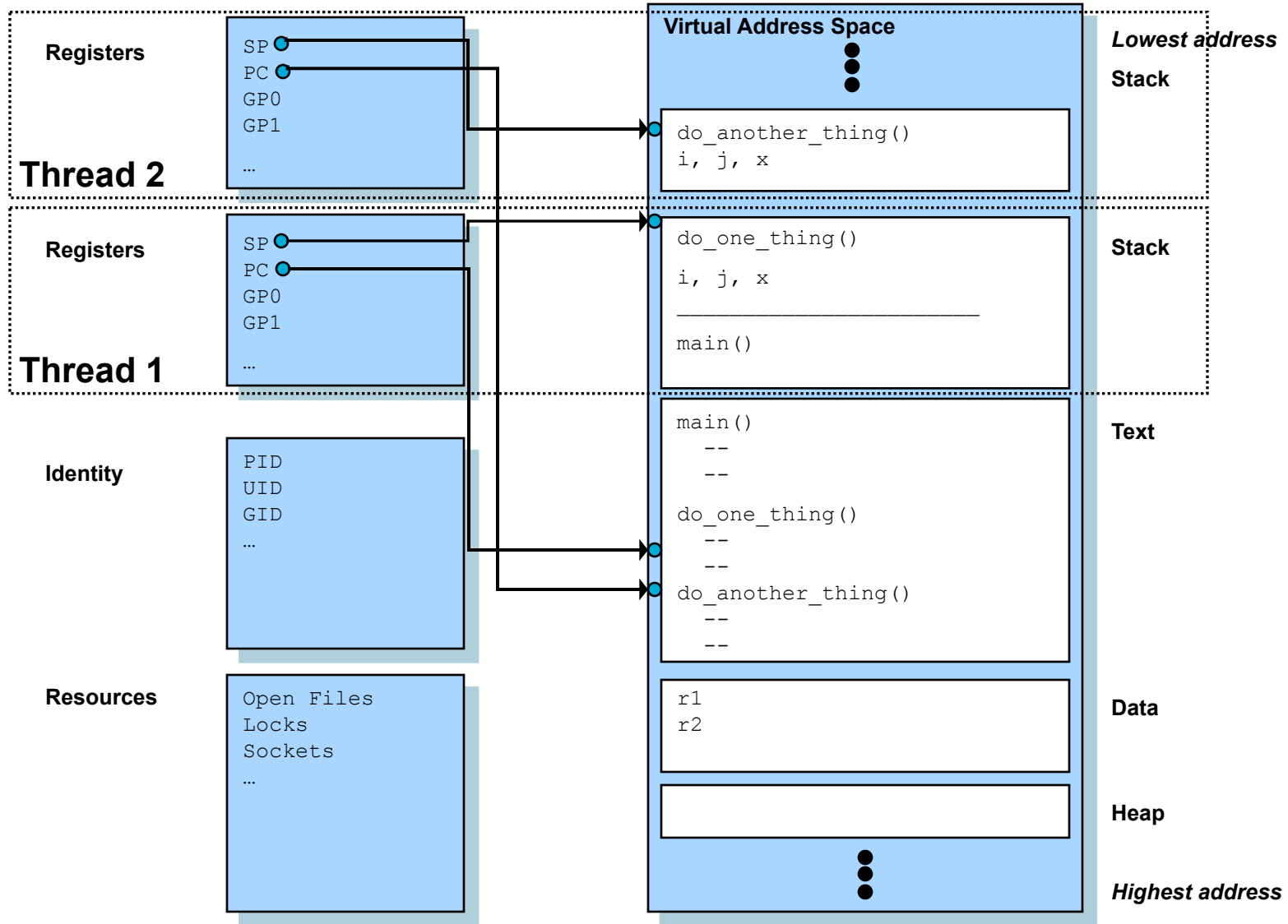
void do_another_thing(int *ptimes)
{
    int i, j, x;

    for (i = 0; i < 4; i++) {
        printf("doing another\n");
        for (j = 0; j < 1000; j++)
            x = x + i;
        (*ptimes)++;
    } /* do_another_thing! */
```

```
void do_wrap_up(int one, int
    another)
{
    int total;
    total = one + another;
    printf("wrap up: one %d, another
        %d and total %d\n", one,
        another, total);
}

int main (int argc, char *argv[])
{
    do_one_thing(&r1);
    do_another_thing(&r2);
    do_wrap_up(r1,r2);
    return 0;
} /* main! */
```

Layout in memory & threading



Threads libraries

- Pthreads – POSIX standard (IEEE 1003.1c)
 - API specifies behavior of the thread library, implementation is up to the developers of the library
 - Common in UNIX OSs (Solaris, Linux, Mac OS X)

```
▪ int pthread_create(pthread_t *restrict thread,  
    const pthread_attr_t *restrict attr,  
    void *(*start_routine) (void*),  
    void *restrict arg);  
  
▪ void pthread_exit(void *value_ptr);  
  
▪ int pthread_join(pthread_t thread, void **value_ptr);  
  
▪ int pthread_yield(void);  
  
▪ int pthread_attr_destroy(pthread_attr_t *attr);  
  
▪ int pthread_attr_init(pthread_attr_t *attr);
```

Thread libraries

- ...
- Win32 threads – slightly different (more complex API)
- Java threads
 - Managed by the JVM
 - May be created by
 - Extending Thread class
 - Implementing the Runnable interface
 - Implementation model depends on OS (1-to-1 in Windows but many-to-many in early Solaris)

Multithreaded C/POSIX

```
/* shared by thread(s) */
int sum;

/* runner: the thread */
void *runner(void *param)
{
    int i, upper = atoi(param);

    sum = 0;
    for (i = 1; i < upper; i++)
        sum += 1;
    pthread_exit(0);
} /* runner! */
```

$$sum = \sum_{i=0}^N i$$

```
int main (int argc, char *argv[])
{
    pthread_t tid;    /* thread id */

    /* set of thread attrs */
    pthread_attr_t attr;

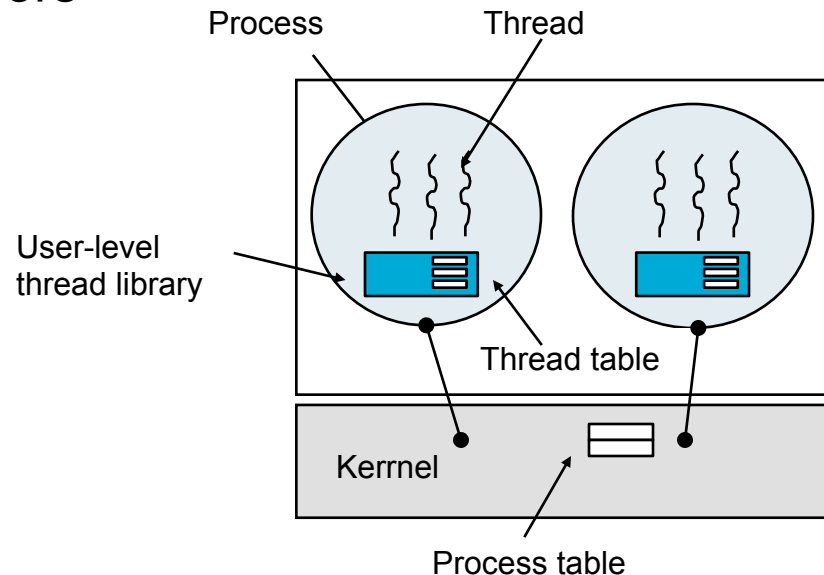
    if (argc != 2 || atoi(argv[1]) < 0) {
        fprintf (stderr, "usage: %s <int>
\n", argv[0]);
        exit(1);
    }

    /* get default attrs */
    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner,
        argv[1]);

    /* wait to exit */
    pthread_join(tid, NULL);
    printf("sum = %d\n", sum);
    exit(0);
} /* main! */
```

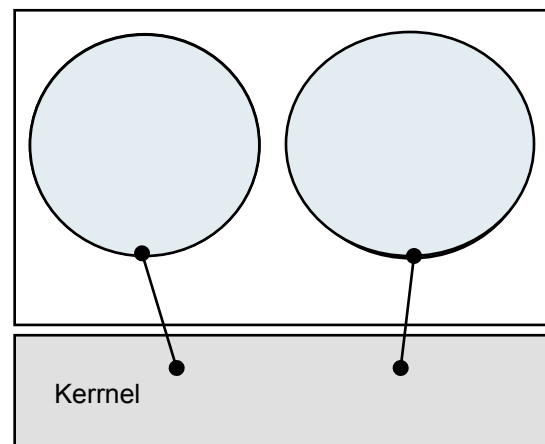

User-level threads

- Kernel unaware of threads – no modification required
- Run-time system or thread manager
 - A collection of procedures
 - No need to manipulate address space (only kernel can do)
- Each process needs its own thread table
 - Run-time system multiplexes user-level threads on top of “virtual processors”



Implementing threads in user-space

- Pros
 - Thread switch is very fast
 - No need for kernel support
 - Customized scheduler
 - Each process ~ virtual processor
- Cons - 'real world' factors
 - Multiprogramming, I/O, Page faults
 - Blocking system calls? Can you check?

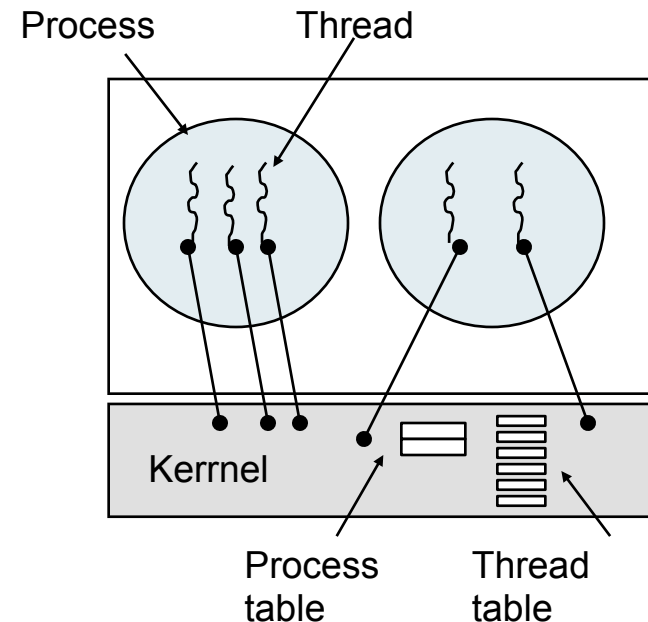


What you see ...

And what the kernel sees ...

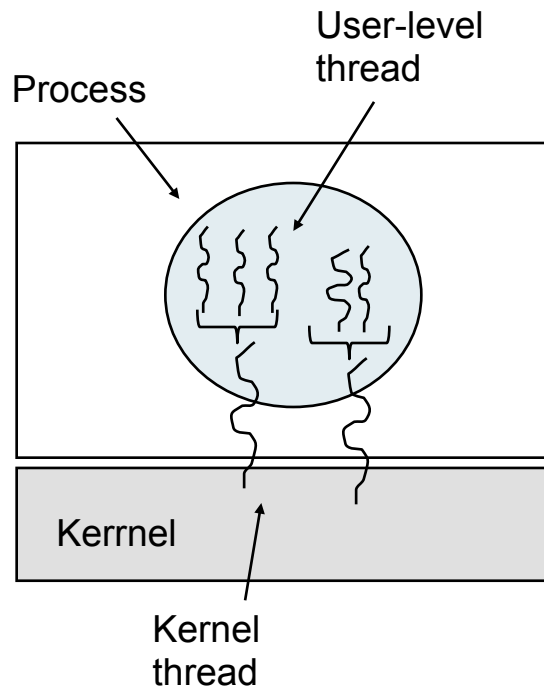
Implementing threads in the kernel

- No need for runtime system
- No wrapper for system calls
- But ... creating threads is more expensive
 - Recycle? Mark a destroy thread as not runnable and reuse it later to save overhead
- And system calls are expensive



Hybrid thread implementations

- Trying to get the best of both worlds
- Multiplexing user-level threads onto kernel-level threads
- One popular variation – two-level model (you can bound a user-level thread to a kernel one)



Processes and threads' performance

- On an old 700MHz Pentium running Linux 2.2.*
- Processes
 - `fork()/exit()` - 251μsec
- Kernel-level thread
 - `pthread_create()/pthread_join()` - 94μsec (2.6x faster)
- User-level thread
 - `pthread_create()/pthread_join()` - 4.5μsec (21x faster)

Scheduler activations*

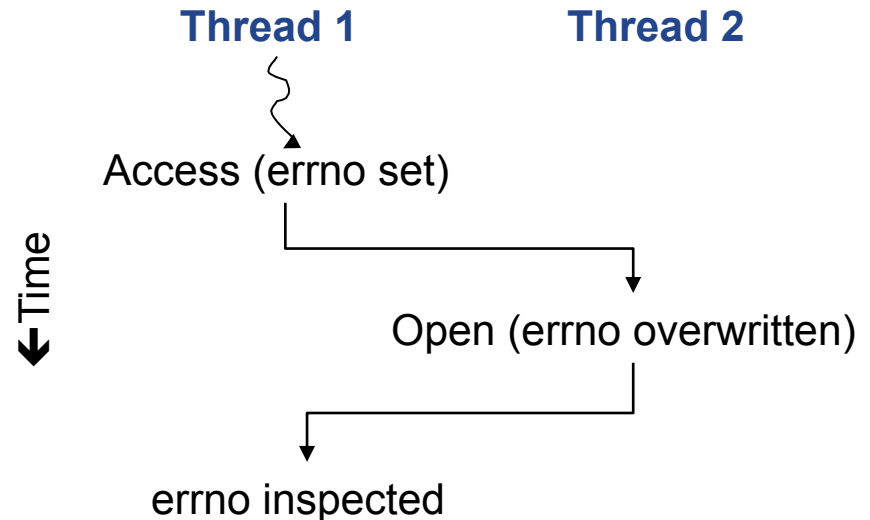
- Goal
 - Functionality of kernel threads &
 - Performance of user-level threads
 - Without special non-blocking system calls
- Problem: needed control & scheduling information distributed bet/ kernel & each app's address space
- Basic idea
 - When kernel finds out a thread is about to block, upcalls the runtime system (activates it at a known starting address)
 - When kernel finds out a thread can run again, upcalls again
 - Run-time system can now decide what to do
- Pros – fast & smart
- Cons – upcalls violate layering approach

Complications with threads ...

- Semantics of `fork()` & `exec()` system calls
 - Duplicate all threads or single-threaded child?
 - Are you planning to invoke `exec()`?
- Other system calls (closing a file, `lseek`, ...?)
- Signal handling, handlers and masking
 1. Send signal to each thread – too expensive
 2. A master thread per process – asymmetric threads
 3. Send signal to an arbitrary thread (control C?)
 4. Use heuristics to pick thread (SIGSEGV & SIGILL – caused by thread, SIGTSTP & SIGINT – caused by external events)
 5. Create a thread to handle each signal – situation specific
- Stack growth

Single-threaded to multithreaded

- Threads and global variables
 - An example problem



- Prohibit global variables? Legacy code?
- Assign each thread its own global variables
 - Allocate a chunk of memory and pass it around
 - Create new library calls to create/set/destroy global variables

Single-threaded to multithreaded

- Many library procedures are not reentrant
- Re-entrant: *able to handle a second call while not done with previous one*
 - e.g. assemble msg in a buffer before sending it
- Solutions
 - Rewrite library?
 - Wrappers for each call?
- None of the problem is a showstopper, just a warning when going from single to multithreaded systems

Summary

- You want multiple threads per address space
- Kernel-level threads are
 - More efficient than processes, but
 - Not cheap; all operations require a kernel call and parameter check
- User-level threads are
 - Really fast
 - Great for common-case operations, but
 - Can suffer in uncommon cases due to kernel obliviousness
- Scheduler activations are a good answer

How things start to go wrong ...

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

static volatile int counter = 0;

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;
}

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");
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

```
~/sandbox$ ./sharedCounter
main: begin (counter = 0)
A: begin
B: begin
B: done
A: done
main: done with both (counter = 20000000)
```

```
~/sandbox$ ./sharedCounter
main: begin (counter = 0)
A: begin
B: begin
B: done
A: done
main: done with both (counter = 11353201)
~/sandbox$ ./sharedCounter
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 11598589)
```

What's wrong?!

Next time

- Synchronization
 - Race condition & critical regions
 - Software and hardware solutions
 - Review of classical synchronization problems
 - ...
- *What really happened on Mars?*
http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html

