

Threads



Today

- Why threads
- Thread model & usage
- Implementing threads
- Scheduler activations
- Making single-threaded code multithreaded

Next time

- CPU Scheduling

The problem with processes

- A process consists of (at least):
 - An address space
 - The code for the running program
 - The data for the running program
 - 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, sound channels, ...
- A lot of concepts bundled together!

The problem with processes

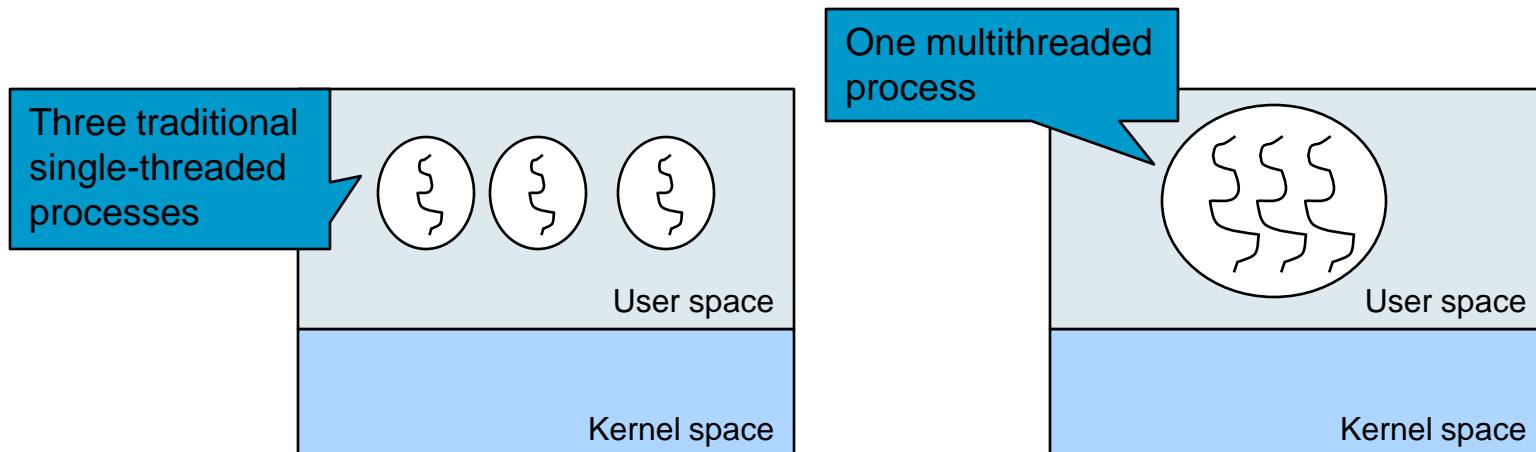
- Many programs need to perform mostly independent tasks that do not need to be serialized
 - e.g. web server, text editor, database server, ...
- In each examples
 - Everybody wants to run the same code
 - ... wants to access the same data
 - ... has the same privileges
 - ... uses the same resources (open files, net connections, etc.)
- But you'd like to have multiple HW execution states:
 - An execution stack & SP
 - PC indicating the next instruction
 - A set of general-purpose processor registers & their values

How can we get this?

- Given the process abstraction as we know it
 - fork several processes
 - cause 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.
- Some 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)

The thread model

- Traditionally
 - Process = 1 address space + 1 thread of execution
 - 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)

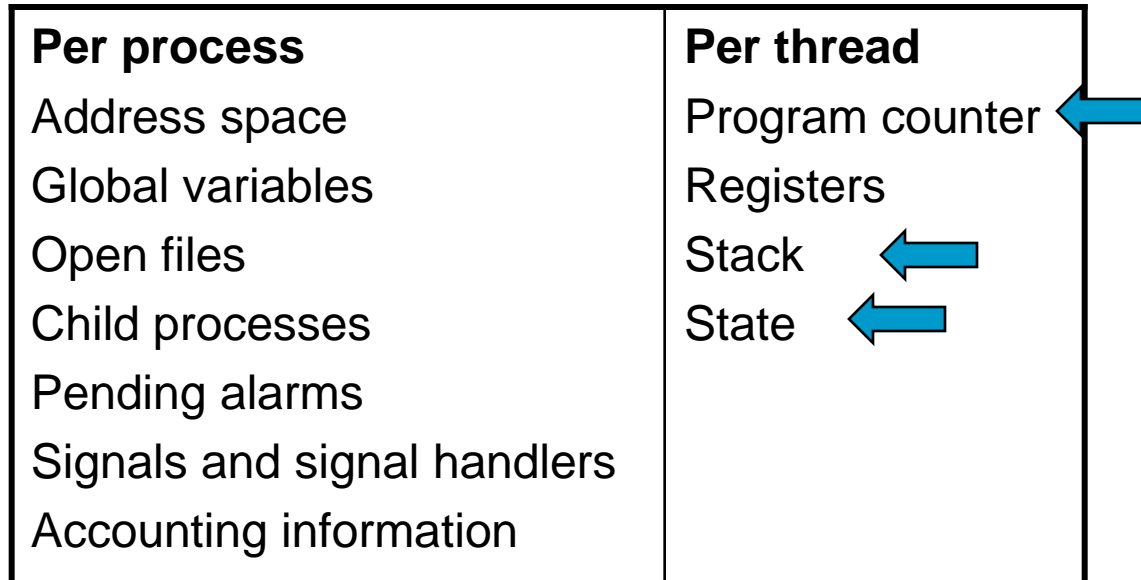


The thread model

- Concurrency & parallelism
 - Concurrency – what's possible with infinite processors
 - Provided at the
 - System level: Kernel recognizes multiple threads of control within a process & schedules them independently
 - Application level: Through user-level thread library; a good structuring tool
 - Parallelism – your actual degree of parallel exec.
- Threads states ~ processes states
- One stack per thread – w/ one frame per procedure called but not yet returned from
- Common calls
 - `thread_create()`
 - `thread_exit()`
 - `thread_wait()`
 - `thread_yield()` (*why would you need this?*)

The thread model

- Share and private items

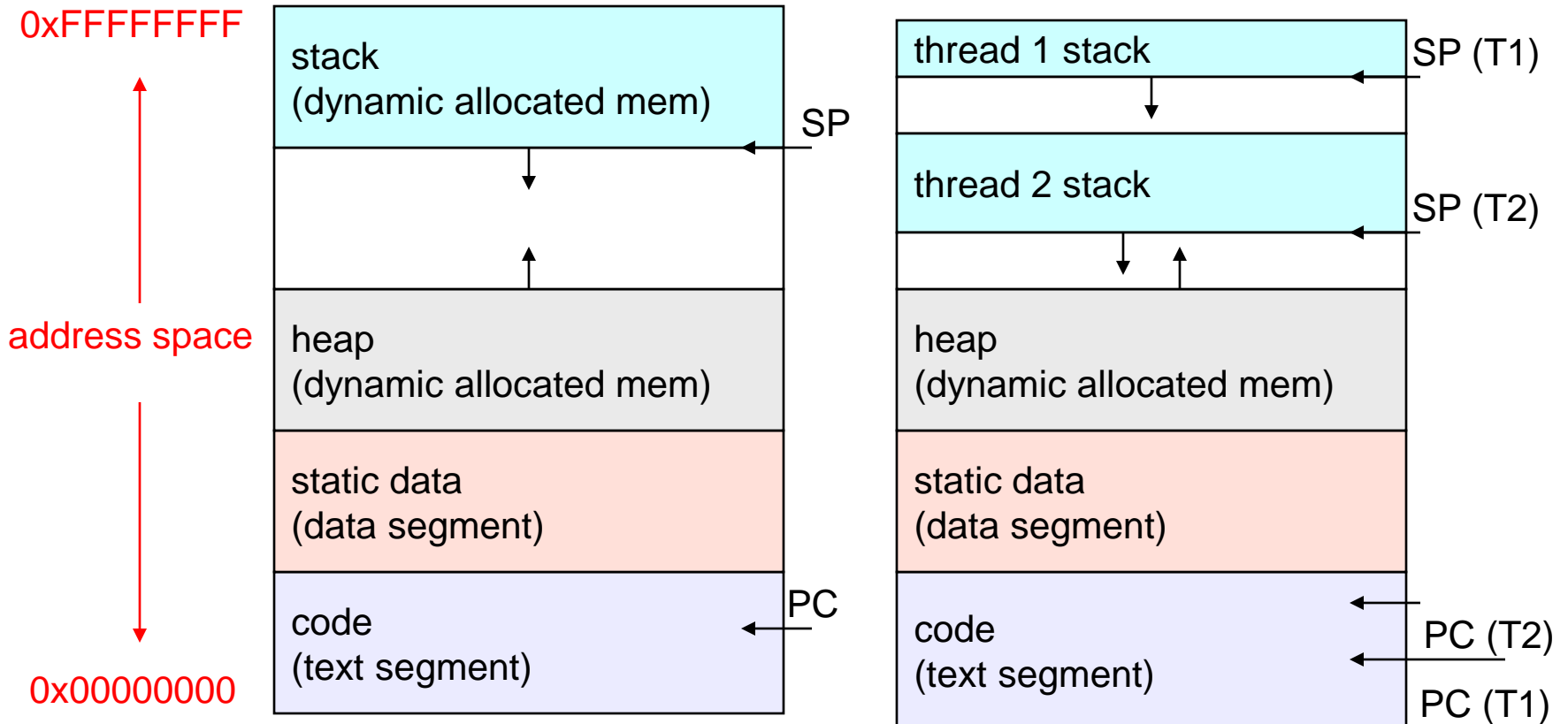


- No protection bet/ threads
(Should they be?)

Old and new process address space

Old one without threads

New one with threads



A simple example

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

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

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

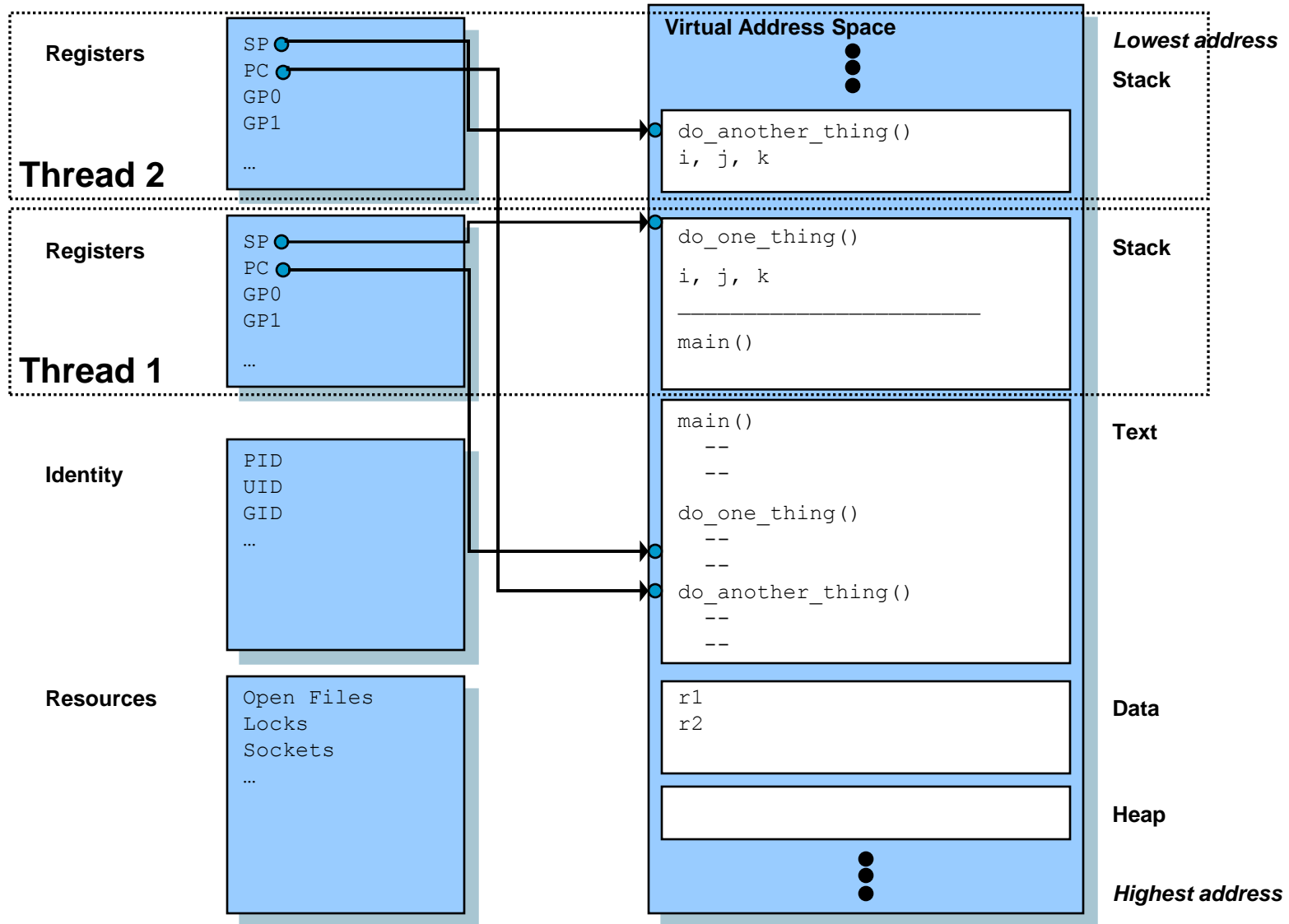
void do_another_thing(int *ptimes)
{
    int i, j, k;

    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



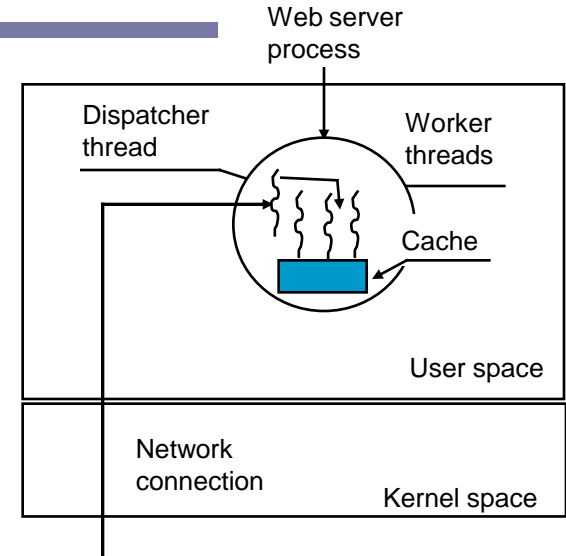
Benefits of threads

- A web server

- Single-threaded: no parallelism, blocking system calls
- Event-driven: parallelism, non-blocking system calls, interrupts
- Multithreaded: parallelism, blocking system calls

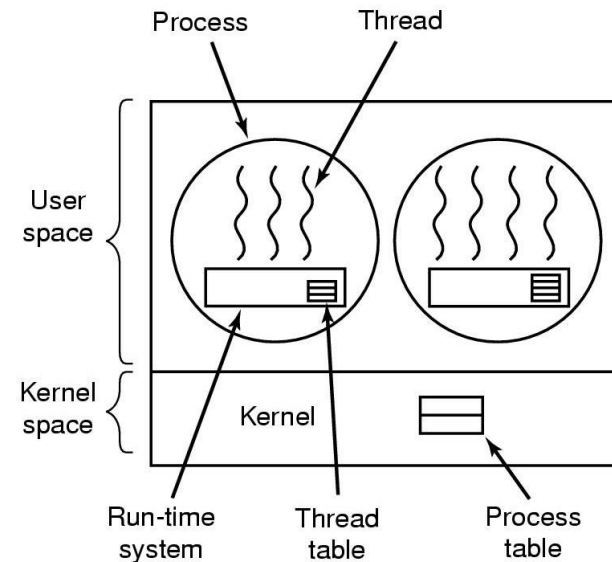
- Reasons for threads

- Simpler programming model when application has multiple, concurrent activities
- Easy/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



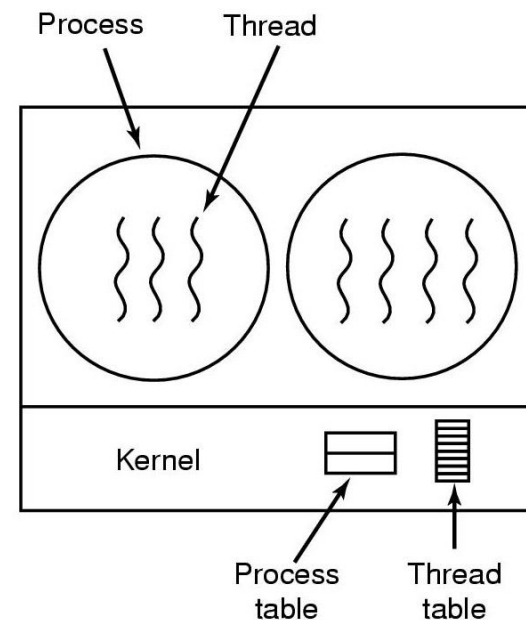
Implementing threads in user space

- Kernel unaware of threads – no modification required (many-to-one model)
- Run-time system: a collection of procedures
- Each process needs its own thread table
- 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?



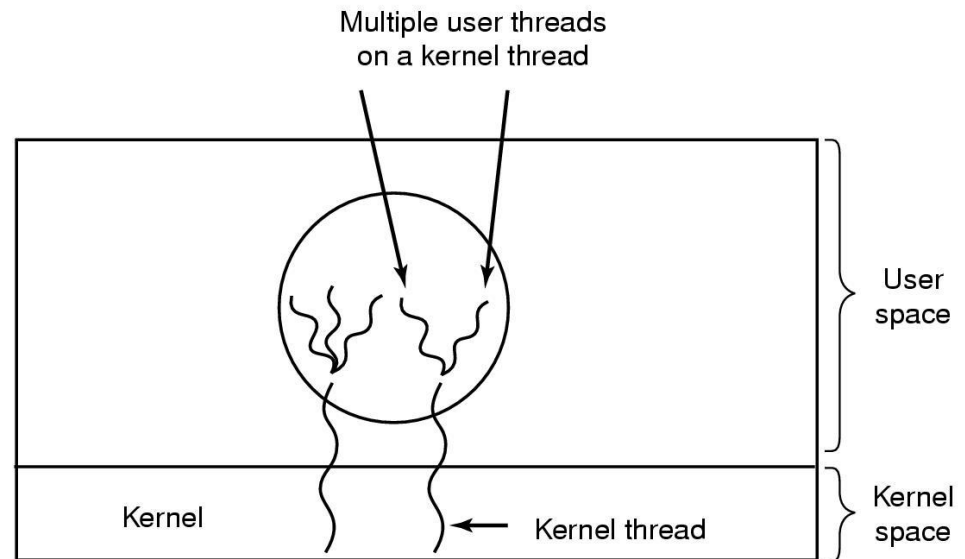
Implementing threads in the kernel

- One-to-one model
- No need for runtime system
- No wrapper for system calls
- But ... creating threads is more expensive – recycle
- And system calls are expensive



Hybrid thread implementations

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



Costs of threads (creation)

Creation time	User-level threads	LWP/Kernel-level threads	Processes
SPARCstation 2, Solaris	52 μ sec	350 μ sec	1700 μ sec
700MHz Pentium, Linux 2.2.*	4.5 μ sec create/join	94 μ sec create/join	251 μ sec fork/exit

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

*Anderson et al., "Scheduler Activations: effective Kernel Support for the User-level Management of Parallelism," SOSP, Oct. 1991.

Thread libraries

- Pthreads – POSIX standard (IEEE 1003.1c) API for thread creation & synchronization
 - 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)
- 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! */
```

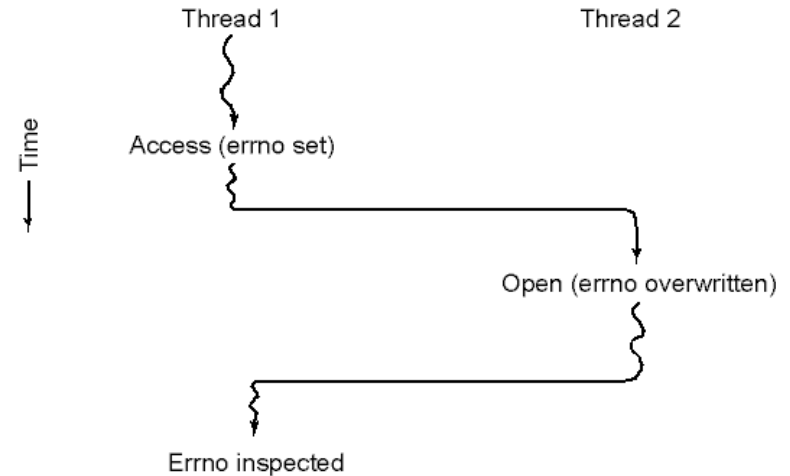
Complications with threads

- 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
- Visibility of threads
- 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?
- Signal handling

Summary

- You really want multiple threads per address space
- Kernel threads are more efficient than processes, but they're still not cheap
 - all operations require a kernel call and parameter verification
- 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
- Next time
 - Multiple processes in the ready queue, but only one processor ... which you should you pick next?