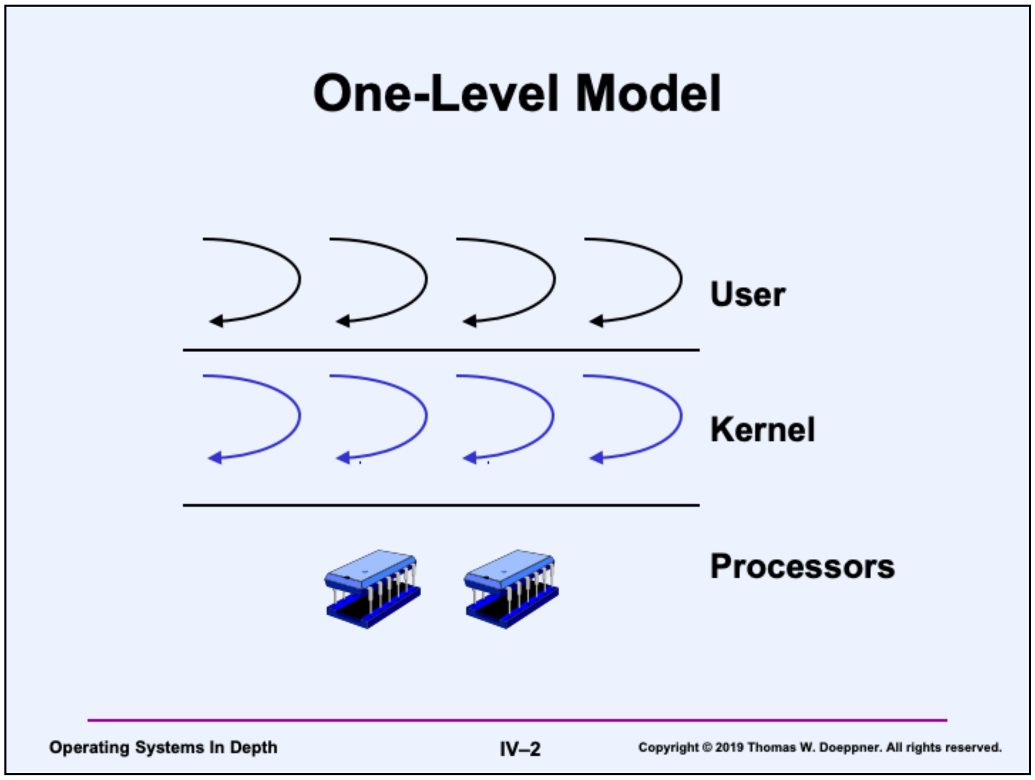
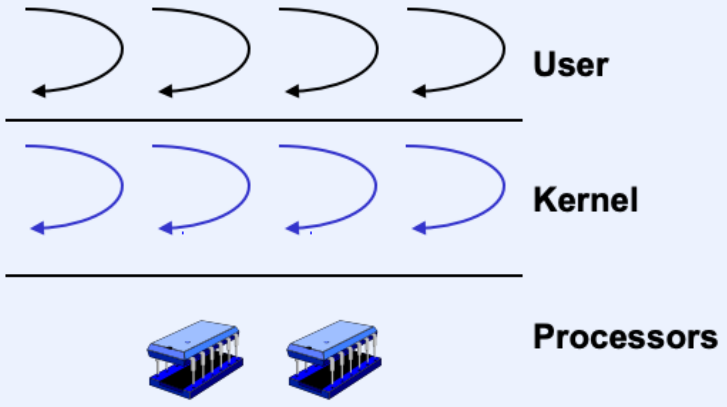
Implementing Threads 3

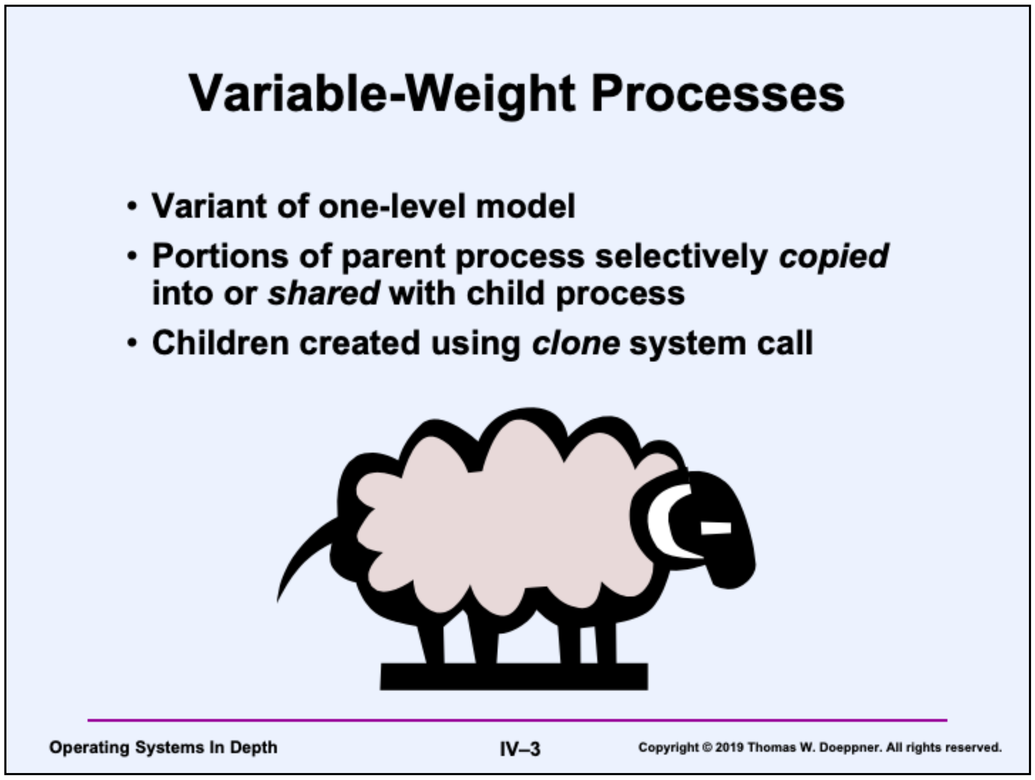


One-Level Model



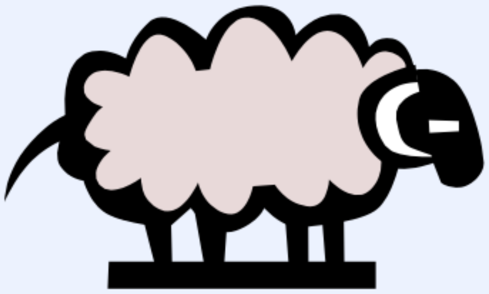
In most systems there are actually two components of the execution context: the user context and the kernel context. The former is for use when an activity is executing user code; the latter is for use when the activity is executing kernel code (on behalf of the chore). How these contexts are manipulated is one of the more crucial aspects of a threads implementation.

The conceptually simplest approach is what is known as the one-level model: each thread consists of both contexts. Thus a thread is scheduled to an activity and the activity can switch back and forth between the two types of contexts. A single scheduler in the kernel can handle all the multiplexing duties. The threading implementation in Windows is (mostly) done this way.



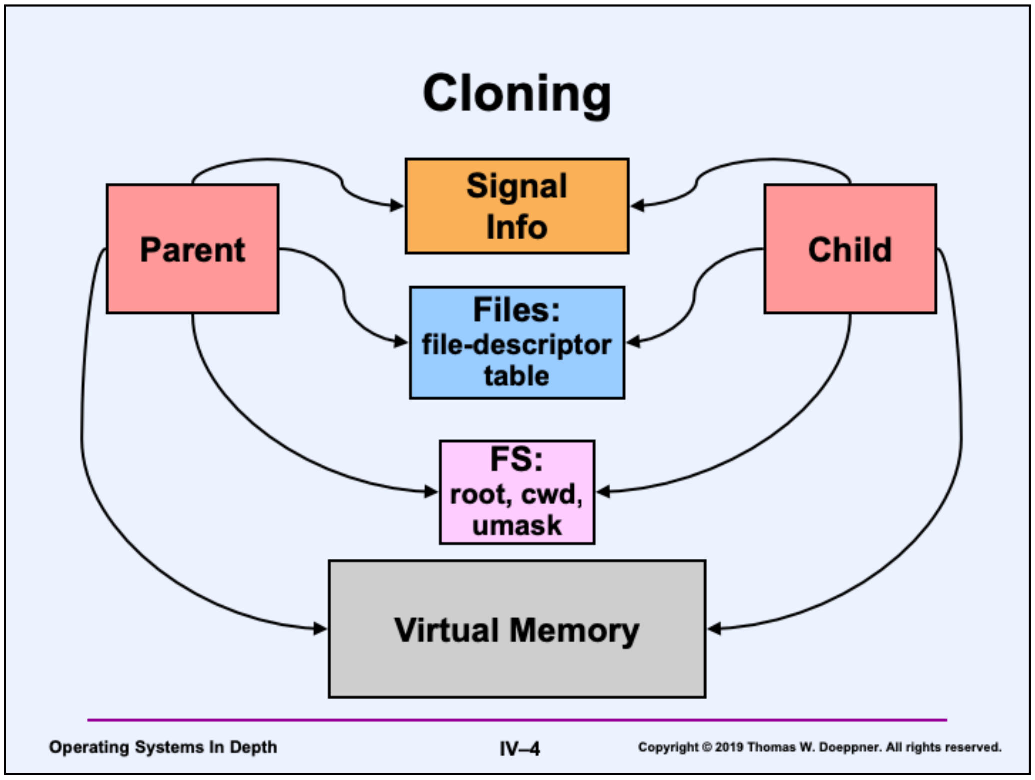
Variable-Weight Processes

- Variant of one-level model
- Portions of parent process selectively *copied* into or *shared* with child process
- Children created using *clone* system call



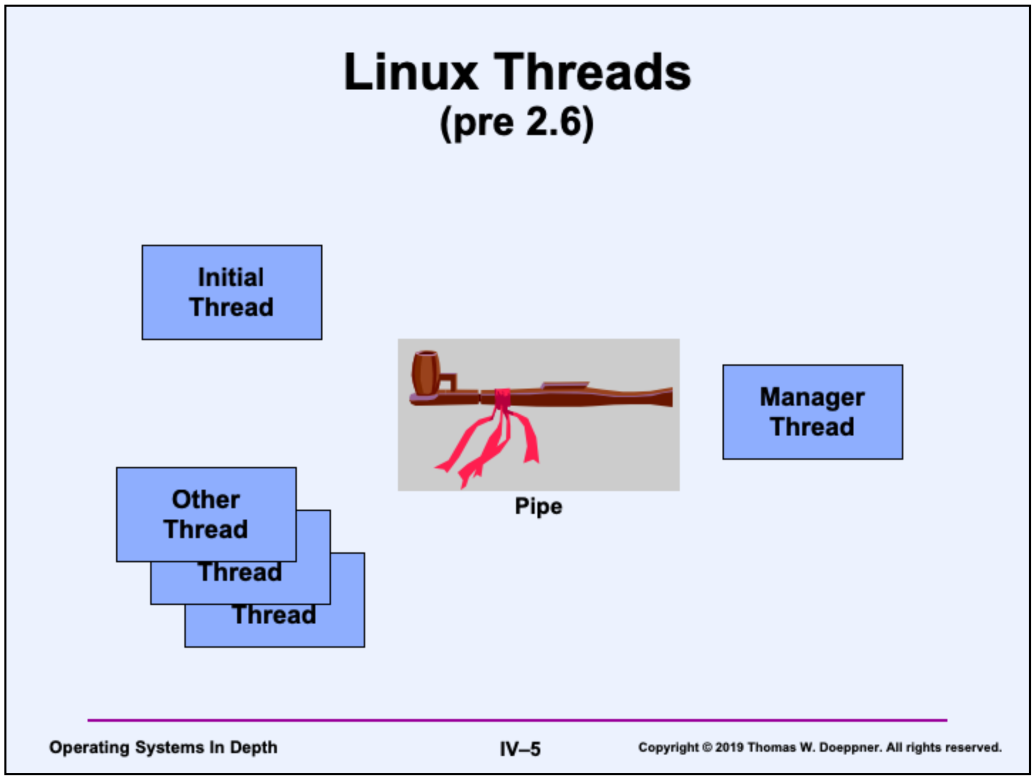
Unlike most other Unix systems, which make a distinction between processes and threads, allowing multithreaded processes, Linux maintains the one-thread-per-process approach. However, so that we can have multiple threads sharing an address space, Linux supports the *clone* system call, a variant of *fork*, via which a new process can be created that shares resources (in particular, its address space) with the parent. The result is a variant of the one-level model.

This approach is not unique to Linux. It was used in SGI's IRIX and was first discussed in early '89, when it was known as *variable-weight processes*. (See "Variable-Weight Processes with Flexible Shared Resources," by Z. Aral, J. Bloom, T. Doeppner, I. Gertner, A. Langerman, G. Schaffer, *Proceedings of Winter 1989 USENIX Association Meeting*.)



Cloning

As implemented in Linux, a process may be created with the *clone* system call (in addition to using the *fork* system call). One can specify, for each of the resources shown in the slide, whether a copy is made for the child or the child shares the resource with the parent. Only two cases are generally used: everything is copied (equivalent to fork) or everything is shared (creating what we ordinarily call a thread, though the "thread" has a separate process ID).



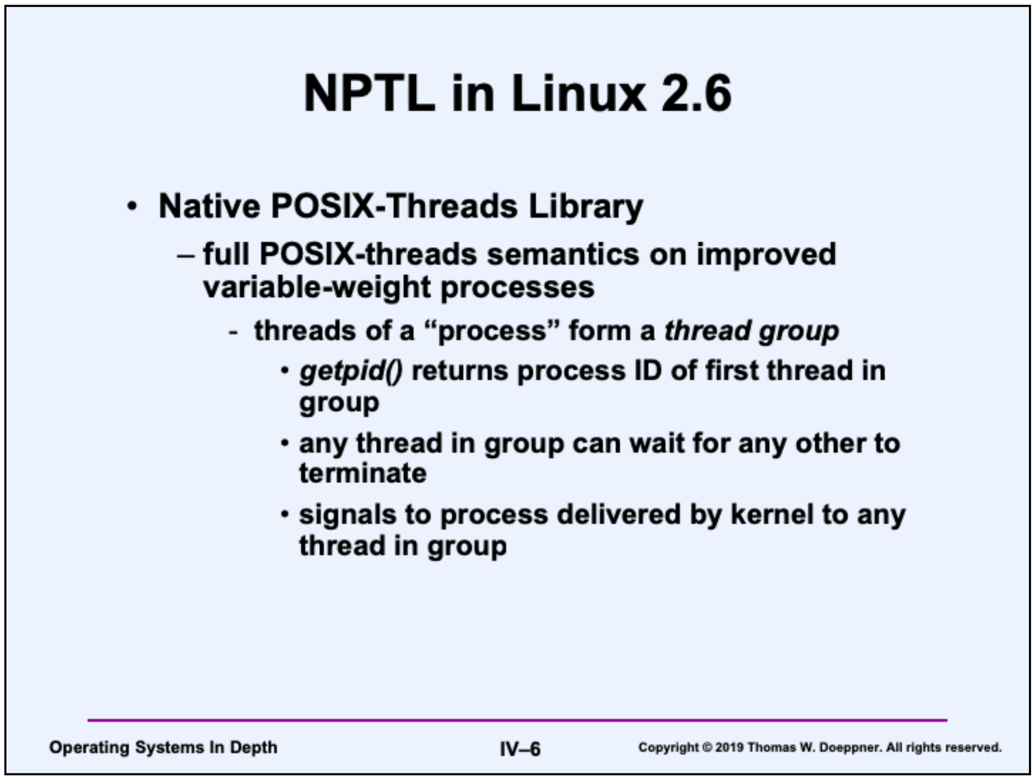
Linux Threads
(pre 2.6)

Building a POSIX-threads implementation on top of Linux's variable-weight processes requires some work. What's discussed here is the approach used prior to Linux 2.6.

Each thread is, of course, a process; all threads of the same computation share the same address space, open files, and signal handlers. One might expect that the implementation of *pthread_create* would be a simple call to clone. This, unfortunately, wouldn't allow an easy implementation of operations such as *pthread_join*: a Unix process may wait only for its children to terminate; a POSIX thread can join with any other joinable thread. Furthermore, if a Unix process terminates, its children are inherited by the init process (process number 1). So that *pthread_join* can be implemented without undue complexity, a special manager thread (actually a process) is the parent/creator of all threads other than the initial thread. This manager thread handles thread (process) termination via the wait4 system call and thus provides a means for implementing *pthread_join*. So, when any thread invokes *pthread_create* or *pthread_join*, it sends a request to the manager via a pipe and waits for a response. The manager handles the request and wakes up the caller when appropriate.

The state of a mutex is represented by a bit. If there are no competitors for locking a mutex, a thread simply sets the bit with a compare-and-swap instruction (allowing atomic testing and setting of the mutex's state bit). If a thread must wait for a mutex to be unlocked, it blocks using a *sigsuspend* system call, after queuing itself to a queue headed by the mutex. A thread unlocking a mutex wakes up the first waiting thread by sending it a Unix signal (via the kill system call). The wait queue for condition variables is implemented in a similar fashion.

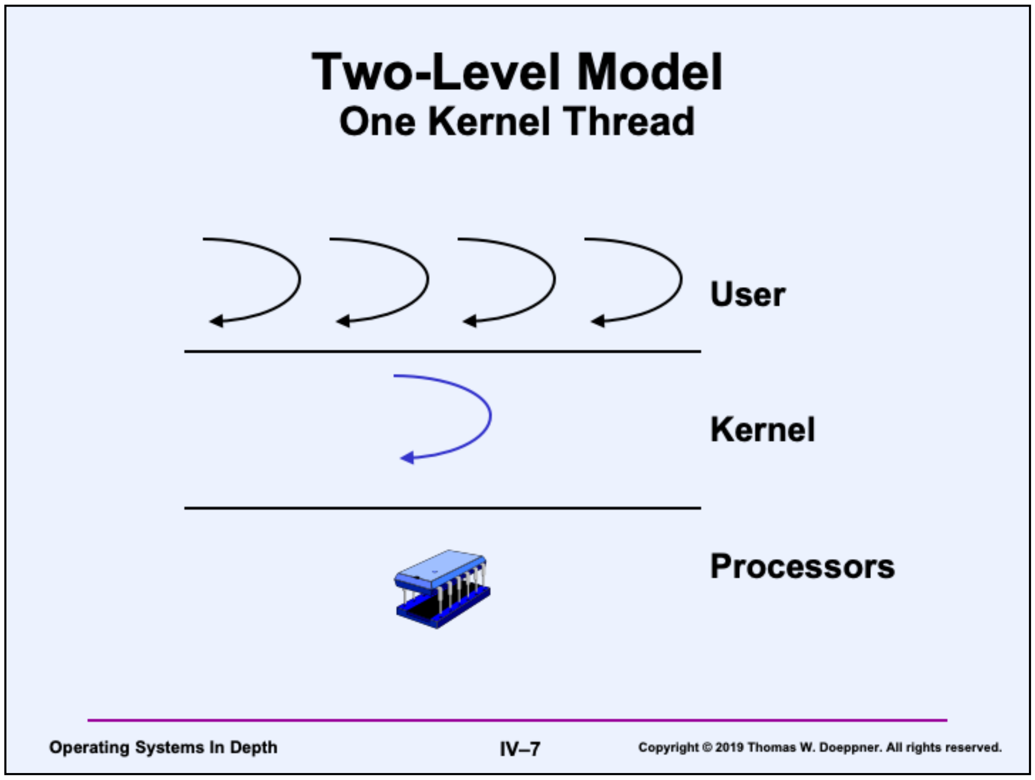
On multiprocessors, for mutexes that are neither recursive nor error-checking, waiting is implemented with an adaptive strategy: under the assumption that mutexes are typically not held for a long period of time, a thread attempting to lock a locked mutex "spins" on it for up to a short period of time, i.e., it repeatedly tests the state of the mutex in hopes that it will be unlocked. If the mutex does not become available after the maximum number of tests, then the thread finally blocks by queuing itself and calling *sigsuspend*.



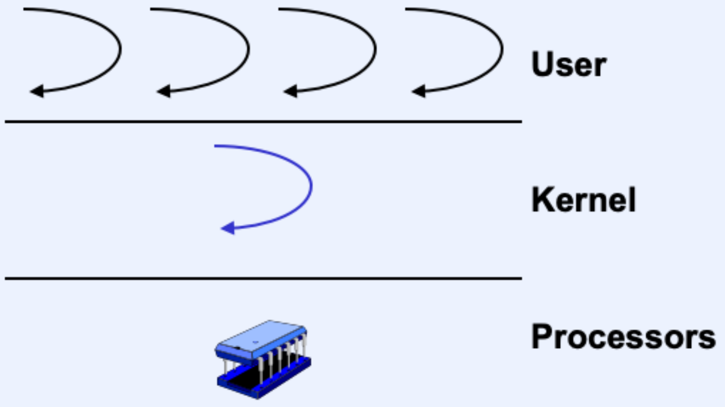
NPTL in Linux 2.6

- Native POSIX-Threads Library
  - full POSIX-threads semantics on improved variable-weight processes
    - threads of a "process" form a *thread group*
      - *getpid()* returns process ID of first thread in group
      - any thread in group can wait for any other to terminate
      - signals to process delivered by kernel to any thread in group

NPTL, the "Native POSIX Threads Library" that comes with most Linux 2.6 systems, provides a big improvement over the previous version of threads on Linux, which is referred to as "Linux Threads." There's no need for a manager thread anymore, signal-handling semantics are now as they should be in POSIX, and synchronization constructs are implemented much more efficiently than on Linux Threads.



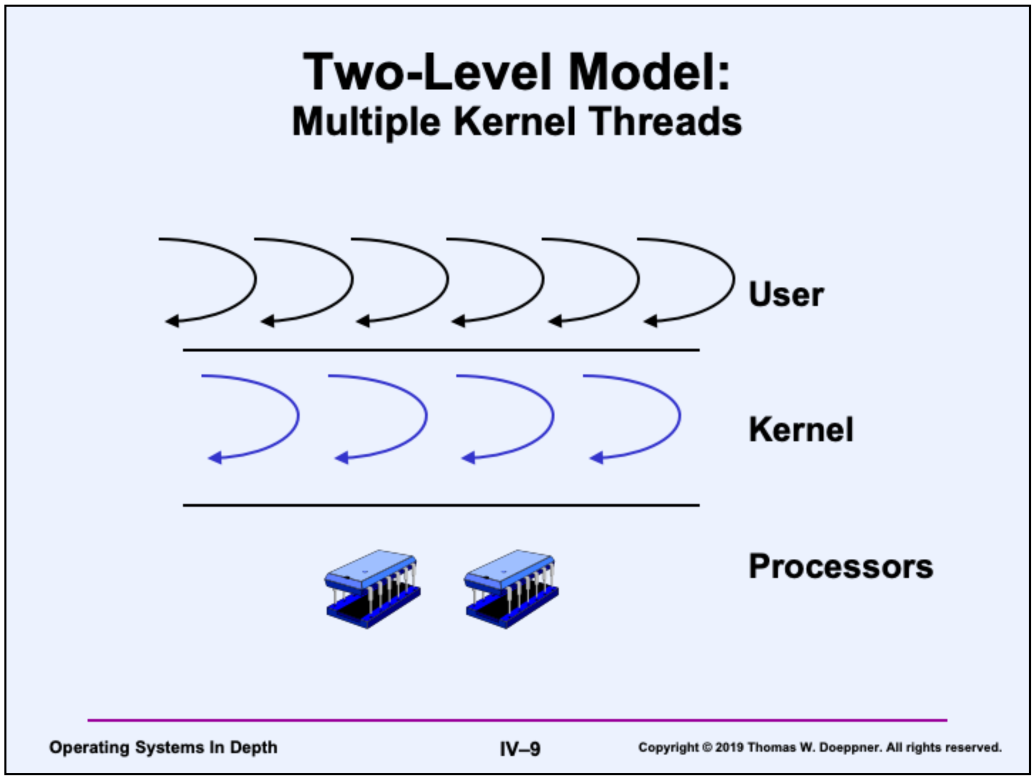
Two-Level Model
One Kernel Thread



Another approach, the two-level model, is to represent the two contexts as separate types of threads: user threads and kernel threads. Kernel threads become “virtual activities” upon which user threads are scheduled. Thus two schedulers are used: kernel threads are multiplexed on activities by a kernel scheduler; user threads are multiplexed on kernel threads by a user-level scheduler. An extreme case of this model is to use only a single kernel thread per process (perhaps because this is all the operating system supports). The Unix implementation of the Netscape web browser was based on this model (recent Solaris versions use the native Solaris implementation of threads), as were early Unix threads implementations. There are two obvious disadvantages of this approach, both resulting from the restriction of a single kernel thread per process: only one activity can be used at a time (thus a single process cannot take advantage of a multiprocessor) and if the kernel thread is blocked (e.g., as part of an I/O operation), no user thread can run.

Coping …

```
ssize_t read(int fd, void *buf, size_t count) {
  ssize_t ret;
  while (1) {
    if ((ret = real_read(fd, buf, count)) == -1) {
      if (errno == EWOULDBLOCK) {
        sem_wait(&FileSemaphore[fd]);
        continue;
      }
    }
    break;
  }
  return(ret);
}
```



Two-Level Model:
Multiple Kernel Threads

A more elaborate use of the two-level model is to allow multiple kernel threads per process. This deals with both the disadvantages described above and is the basis of the Solaris implementation of threading. It has some performance issues; in addition the notion of multiplexing user threads onto kernel threads is very different from the notion of multiplexing threads onto activities—there is no direct control over when a chore is actually run by an activity. From an application's perspective, it is sometimes desired to have direct control over which chores are currently being run.

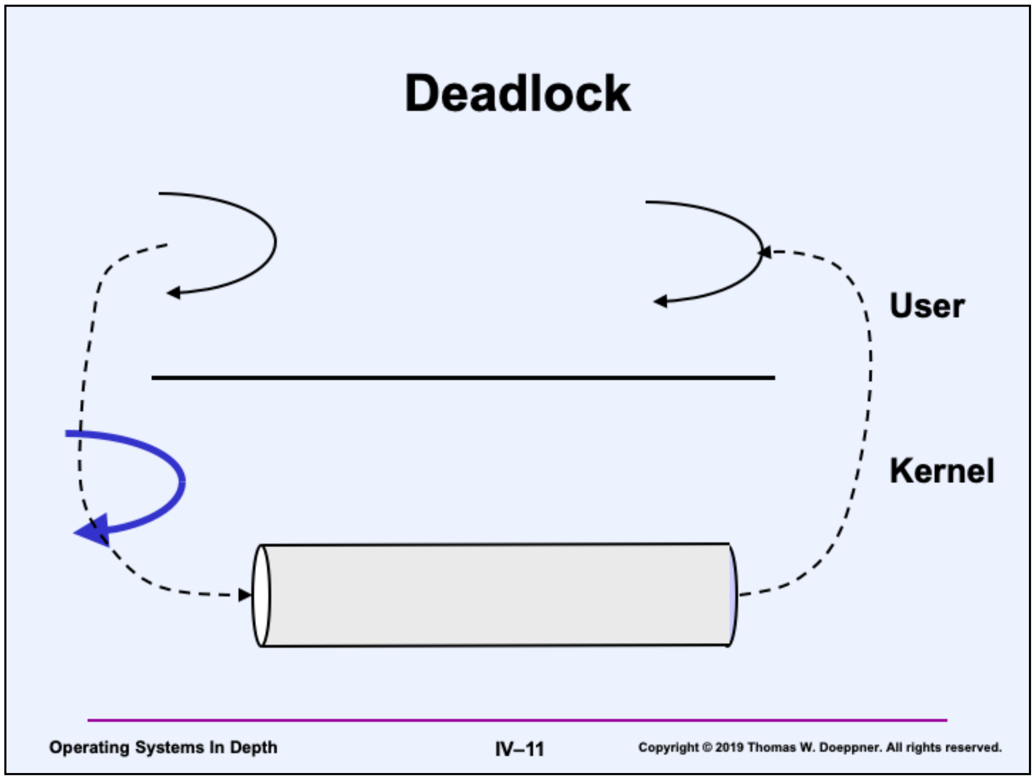
Quiz

One kernel thread for each user thread is clearly a sufficient number of kernel threads in the two-level model. Is it necessary?

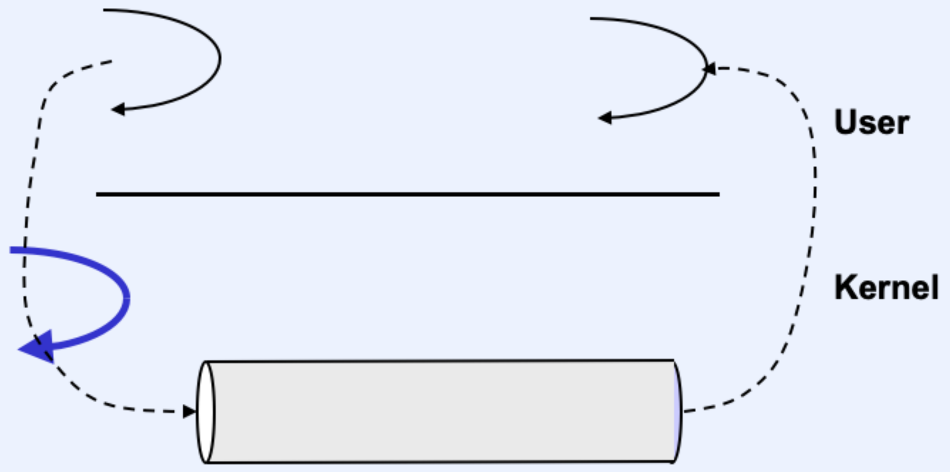
a) there must always be that number of kernel threads for the two-level model to work well.

b) there are situations in which that number is necessary, but they occur rarely.

c) there are no situations in which that number of threads is necessary, as long as there are at least as many kernel threads as processors.



Deadlock



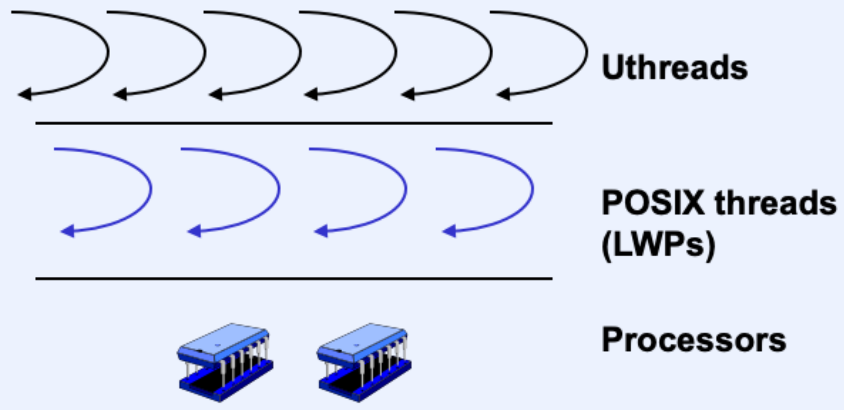
One negative aspect of the two-level model is that its use might induce deadlock. For example, suppose we have two user threads and one kernel thread. One thread is writing into a pipe (using the write system call). However, at the moment the pipe is full. The the call to write blocks. The other user thread is ready to do a read system call on the pipe, thus making it not full and unblocking the first thread, but since there's only one kernel thread and it's blocked (since it's running the first user thread), the second user thread can't read from the pipe and thus we're stuck: deadlocked.

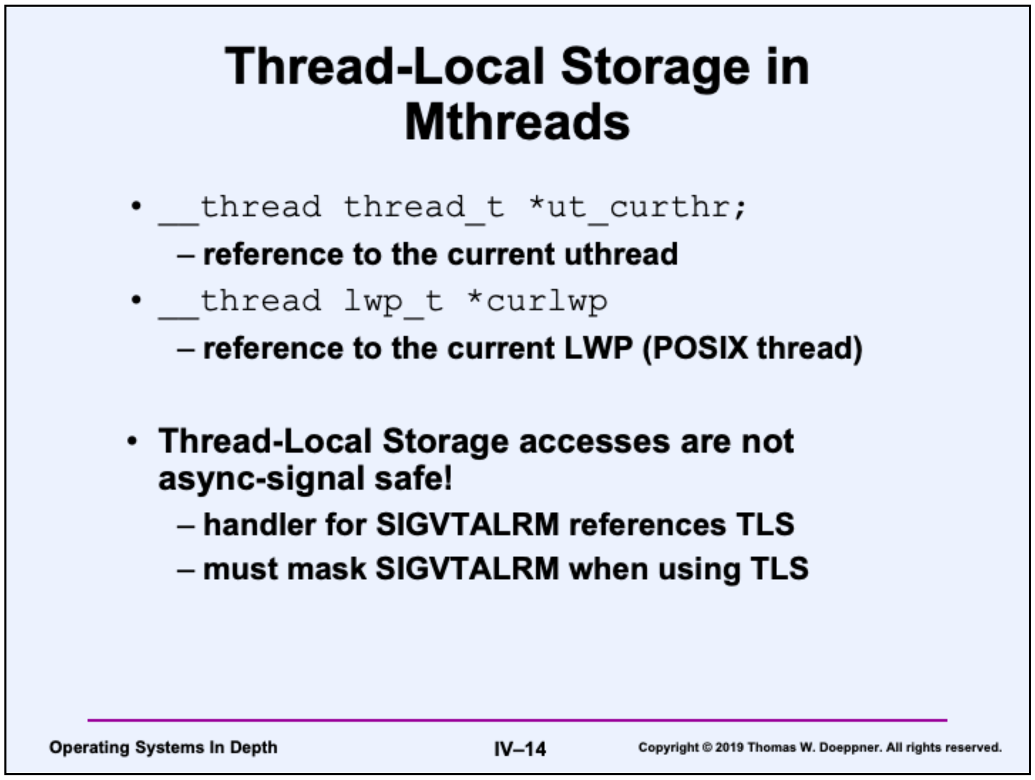
The solution would be to introduce an additional kernel thread if such a situation happens. This was done in the Solaris implementation of the two-level thread model: if all kernel threads in a process are blocked, a new one was automatically created.

MThreads

- Two-level threads implementation of Uthreads
  - kernel-supported threads are POSIX threads
  - user threads based on your implementation of Uthreads
- Effectively a multiprocessor implementation
  - use POSIX mutexes rather than spin locks
  - use POSIX condition variables rather than the idle loop

Two-Level Model:
MThreads



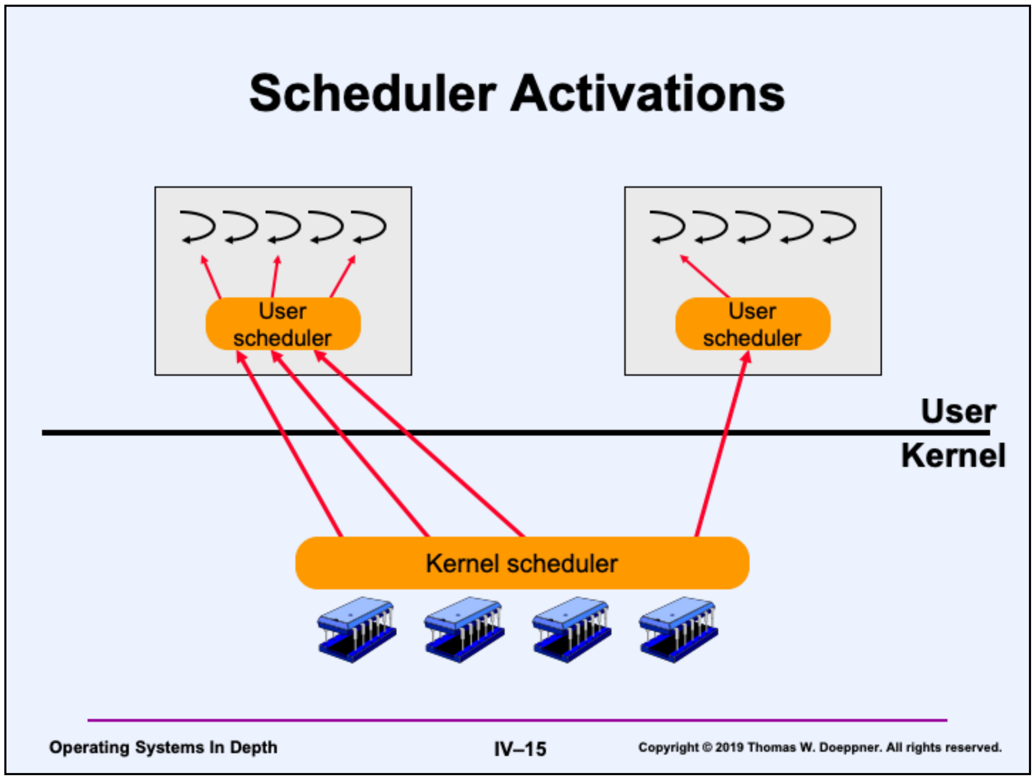


Thread-Local Storage in Mthreads

- `__thread thread_t *ut_curthr;`
  - **reference to the current uthread**
- `__thread lwp_t *curlwp`
  - **reference to the current LWP (POSIX thread)**

- **Thread-Local Storage accesses are not async-signal safe!**
  - **handler for SIGVTALRM references TLS**
  - **must mask SIGVTALRM when using TLS**

Thread-local storage (TLS) is implemented as part of POSIX threads. We use it in mthreads for references to the current utthread and the current LWP. Unfortunately, referencing TLS is not async-signal safe. Since you will be using TLS within the signal handler for SIGVTALRM, make sure that you mask SIGVTALRM when using TLS.


Scheduler Activations
User scheduler
User scheduler
User
Kernel
Kernel scheduler
Operating Systems In Depth
IV–15
Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Yet another approach, known historically as the *scheduler activations model*, is that threads represent user contexts, with kernel contexts supplied when needed (i.e., not as a kernel thread, as in the two-level model). User threads are multiplexed on activities by a user-level scheduler, which communicates to the kernel the number of activities needed (i.e., the number of ready user threads). The kernel multiplexes entire processes on activities—it determines how many activities to give each process. This model, which is the basis for the Digital-Unix (now True64 Unix) threading package, certainly gives direct control to the user application over which chores are being run.

To make some sense of this, let's work through an example. A process starts up, containing a single user execution context (and user thread) and a kernel execution context (and kernel thread). Following the dictates of its scheduling policy, the kernel scheduler assigns a processor to the process. If the kernel thread blocks, the process implicitly relinquishes the processor to the kernel scheduler, and gets it back once it unblocks.

Suppose that the user program creates a new thread (and its associated user execution context). If actual parallelism is desired, code in the user-level library notifies the kernel that two processors are desired. When a processor becomes available, the kernel creates a new kernel execution context; using the newly available processor running in the new kernel execution context, it places an upcall (going from system code to user code, unlike a system call, which goes from user code to system code) to the user-level library, effectively giving it the processor. The library code then assigns this processor to the new thread and user execution context.