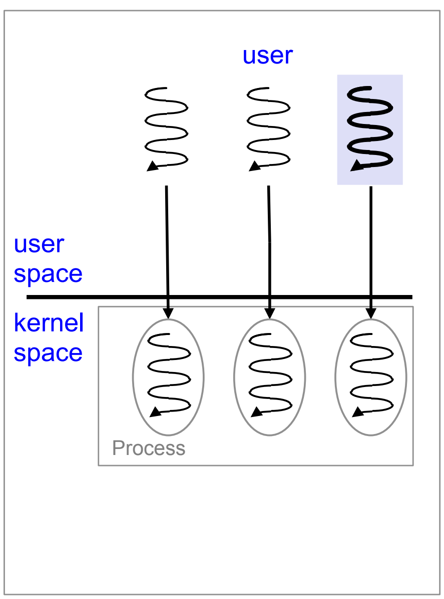
User-Level vs. Kernel-Level Threads

- Kernel-Level Threads
- User-Level Threads
- Advantages and Limitations of User-Level Threads
- Kernel-level Support for User-Level Threads?
- Example: Scheduler Activations

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism". ACM SIGOPS Operating Systems Review, Volume 25, Issue 5, Oct. 1991.

Kernel-Level Threads

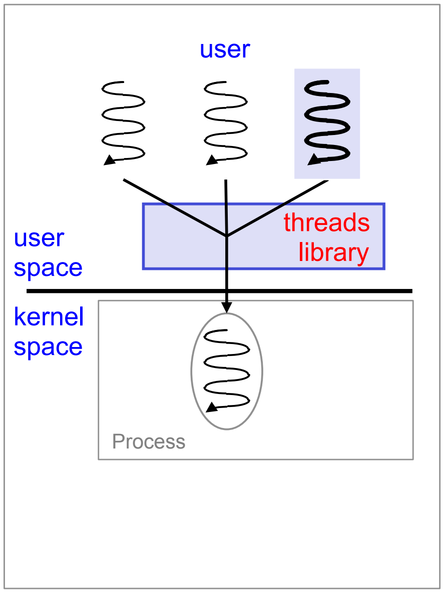


Kernel-Level Threads:
- Threads managed by Operating System

Pros:
- Avoid system-integration issues (see later)

Cons:
- Too heavy-weight!

User-Level Threads



User-Level Threads:

- Managed by runtime library.
- Management operations require no kernel intervention.

User-Level Threads using Standard C Library

```
int getcontext(ucontext_t *ucp);
```

saves current thread's execution context in the structure pointed to by ucp.

ucontext_t: structure type suitable for holding the context for a user thread of execution. It contains at least these fields:

`mcontext_t uc_mcontext;`	saved registers
`stack_t uc_stack;`	stack area
`sigset_t uc_sigmask;`	signals being blocked
`ucontext_t *uc_link;`	context to assume when this one returns

The uc_link field points to the context to resume when this context's entry point function returns. If uc_link is equal to NULL, then the process exits when this context returns.

User-Level Threads using Standard C Library

```
int getcontext(ucontext_t *ucp);
```

saves current thread's execution context in the structure pointed to by ucp.

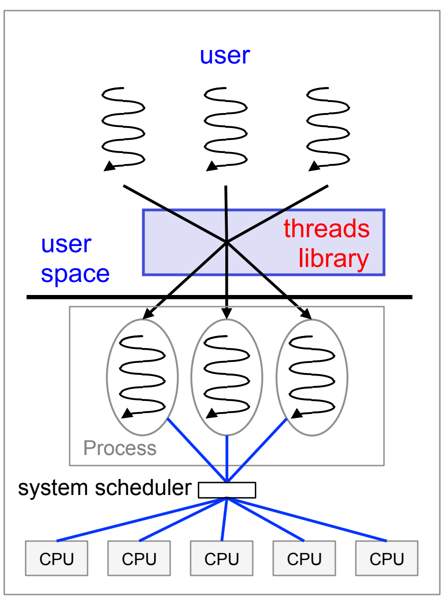
```
int setcontext(const ucontext_t * ucp);
```

make a previously saved thread context the current thread context. The current context is lost and setcontext() does not return.
Instead, execution continue in the context specified by ucp.

```
void makecontext(ucontext_t *ucp, void (*func)(), int argc,
     ...);
```

modifies context pointed to by ucp so that it will continue execution by invoking func() with the arguments provided. Context upc must have previously been initialized by call to getcontext().

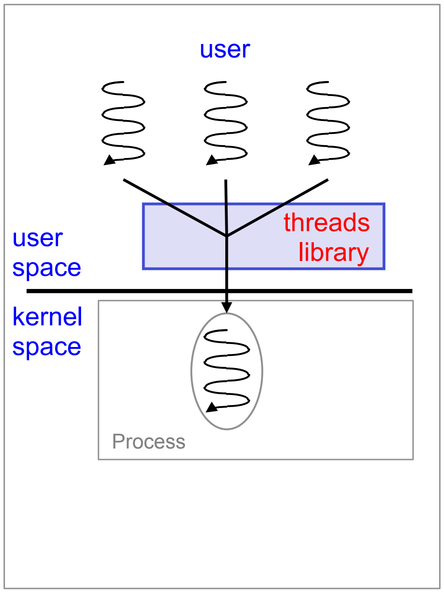
User-Level Threads: in Practice



User-Level Threads:

- Managed by runtime library.
- Management operations require no kernel intervention.

User-Level Threads



User-Level Threads:
- Managed by runtime library.
- Management operations require no kernel intervention.

Pros:
1. Low-cost
2. Flexible (various APIs: POSIX, Actors, Java,...)
3. Implementation requires no change to OS.

Cons:
- Performance issues due to mapping to OS resources (see later)

User-Level Threads: Advantages

Kernel-level threads have inherent **dis**advantages
- **Performance Cost of thread management operations**: Must cross protection boundary on every thread operation, even for operations on threads of the same process.
- **Cost of generality:** A single implementation must be used by all applications. (In contrast, user-level libraries can be tuned to applications.)

Operation	user-level threads	kernel-level threads	kernel-level processes
Null Fork	34	948	11300
Signal-Wait	37	441	1840

Relative latency of thread-operation (Anderson et al.)
(in comparison, procedure call: 7, exception latency: 19)

User-Level Threads: Limitations

User-level threads are difficult to use and to integrate with system services:

1. Kernel events, such as processor preemption, I/0 blocking, and resumption, are handled by the kernel invisibly to the user level.
2. Kernel threads are scheduled obliviously with respect to the user-level thread state.

Scenario:

When a user-level thread blocks, the kernel thread also blocks.

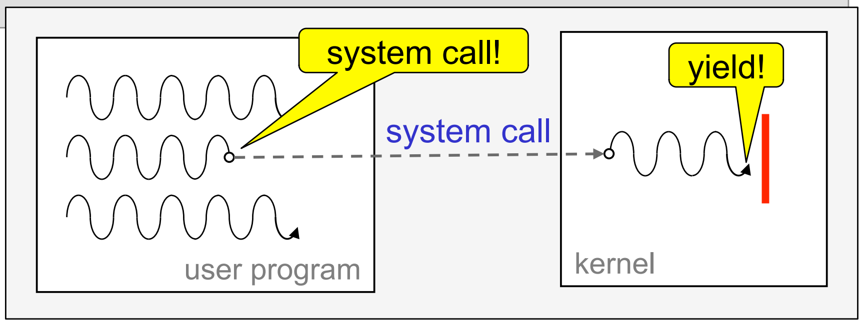
As a result, the physical processor is lost to the process while the blocking request is pending.

User-Level Threads: Limitations

Scenario:

When a user-level thread blocks, the kernel thread serving as its virtual processor also blocks.

As a result, the physical processor is lost to the processes while the blocking request is pending.



User-Level Threads: Limitations

Scenario:

When a user-level thread blocks, the kernel thread serving as its virtual processor also blocks.

As a result, the physical processor is lost to the processes while the blocking request is pending.

Solution (?): Create more kernel threads; when one kernel thread blocks because its user-level thread blocks in the kernel, another kernel thread is available to run user-level threads on that processor.

However: When the thread unblocks, there will be more runnable kernel threads than processors.

-> OS now decides on behalf of the application which user-level threads to run.

User-Level Threads: Limitations

However: When the thread unblocks, there will be more runnable kernel threads than processors.

-> OS now decides on behalf of the application which user-level threads to run.

Solution (?) : The operating system could employ some kind of Round Robin to ensure each thread makes progress.

However: When user-level threads are running on top of kernel threads, round-robin can lead to problems.

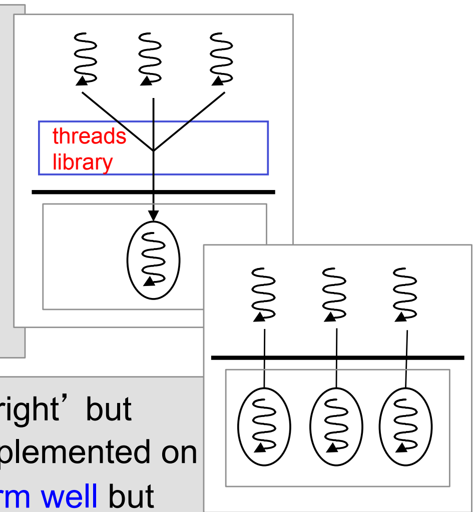
Example: A kernel thread could be preempted while its user-level thread is holding a spin-lock;

Any user-level threads accessing the lock will then spin-wait until the lock holder is re-scheduled.

User-Level or Kernel-Level Threads?

User-Level Threads:
(+) Low-cost, flexible, no changes to OS
(-) Performance issues due to mapping to OS resources

Kernel-Level Threads:
(+) Avoid system integration problems
(-) Too heavyweight



"Dilemma": "Employ kernel threads, which 'work right' but perform poorly, or employ user-level threads implemented on top of kernel threads or processes, which perform well but are functionally deficient." (Anderson *et al.*)

How about Kernel-level Support for ULTs?

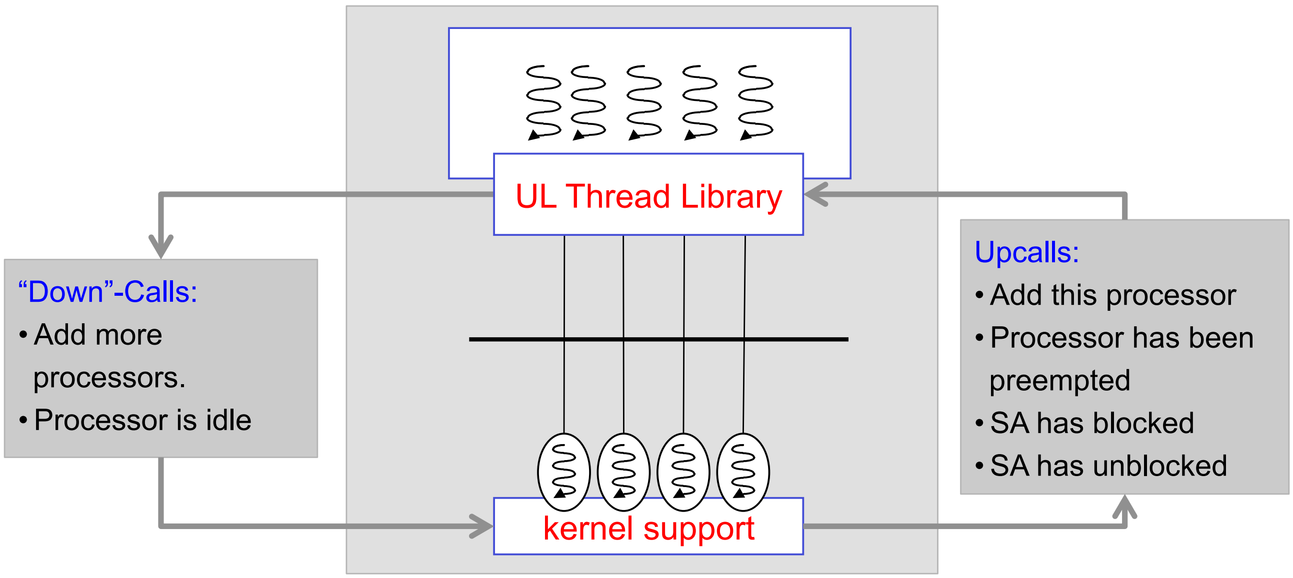
What if we could ...

- ... mimic the behavior of kernel thread management system?
  - i.e., no idling processor in presence of ready threads?
  - i.e., multiprogramming within and across processes?
- ... keep thread management overhead same as user-level threads?

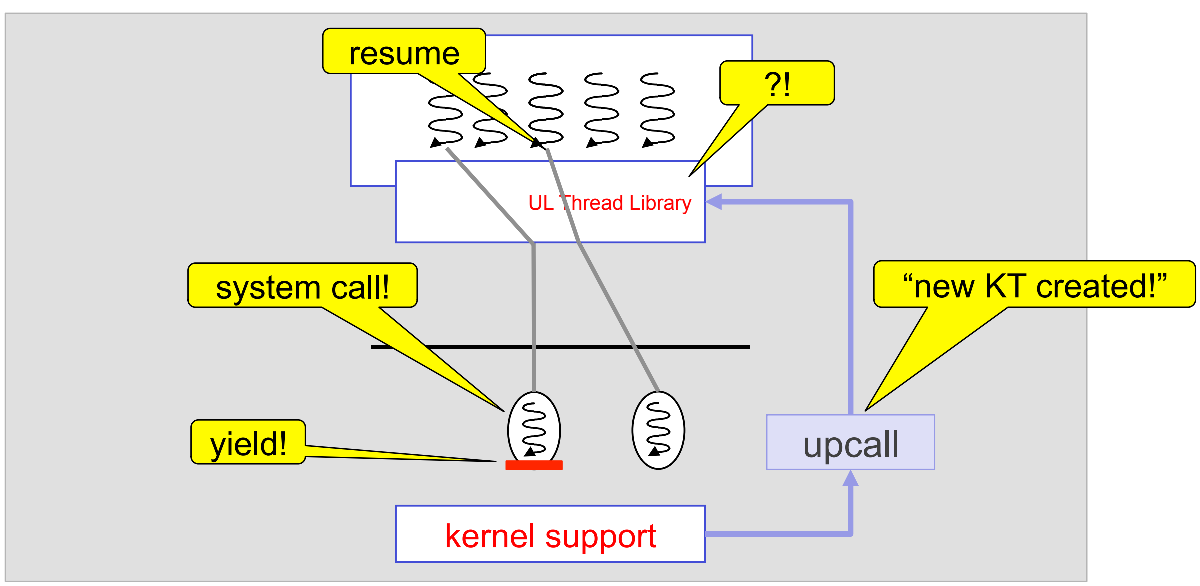
How about ...

- ... kernel allocates physical processors to processes.
- ... UL thread system has complete control over which thread to run on allocated processors.
- ... UL thread system knows about suspended/resumed threads in kernel.
- ... UL thread system can request/release processors.

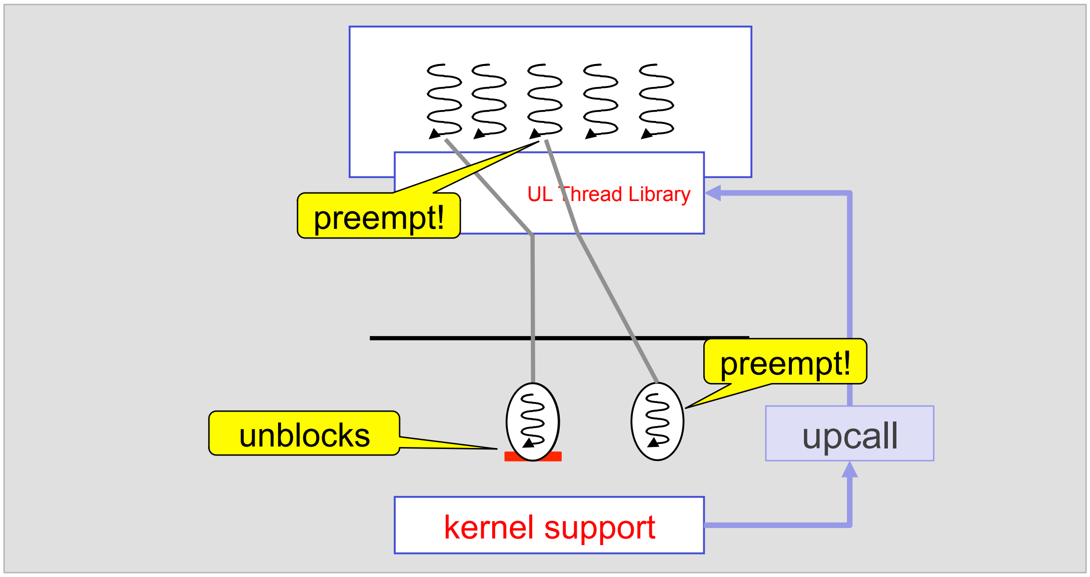
One Solution: "Scheduler Activations"



Handling Blocking Threads using Scheduler Activations



Resuming Blocked Threads using Scheduler Activations



User-Level vs. Kernel-Level Threads

- Kernel-Level Threads
- User-Level Threads
- Advantages and Limitations of User-Level Threads
- Kernel-level Support for User-Level Threads?
- Example: Scheduler Activations