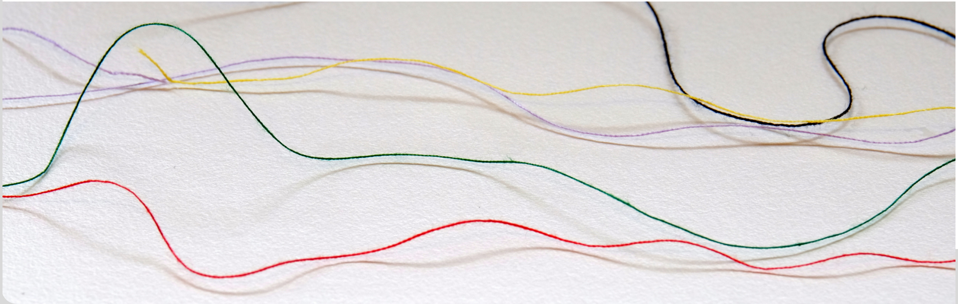
Operating Systems

04. Threads

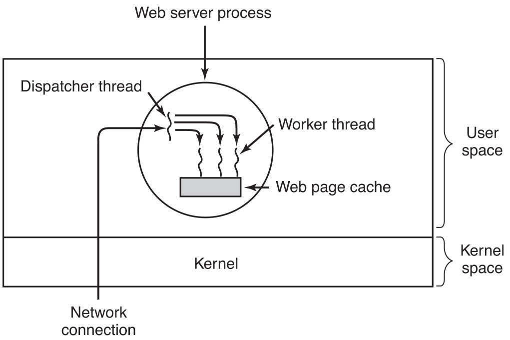
Prof. Dr. Frank Bellosa | WT 2020/2021



Processes vs. Threads

- In traditional OSes, each process has
  - it's own address space (AS)
  - it's own set of allocated resources
  - one thread of execution (one execution state)
- Modern OSes handle processes and threads of execution more flexibly
  - Processes provide the abstraction of an address space and resources
  - Threads provide the abstraction for execution states of that AS
- Research OSes already offer multiple address spaces to different threads of the same process
  - e.g. for live code patching of multi-threaded processes
    see related work [RDF^{+}20]

Why have multiple threads at all?

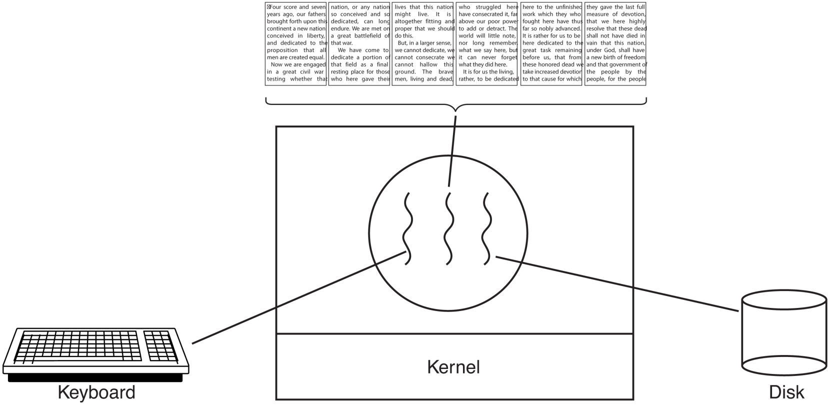


- Many programs do multiple "things" at once
- Multi-threaded web server [TB15]
  - Accept new connections
  - Read request from client
  - Fetch required data
  - Process and deliver data
- Multiple threads can potentially run in parallel on a multiprocessor
- Some of these activities may block

→ Writing a program made of many sequential threads may be easier than avoiding blocking operations (see related work "Threads vs. Events")

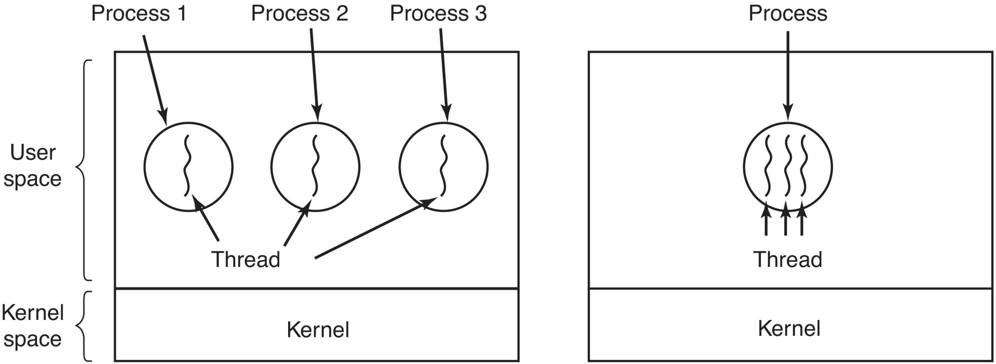
Why have multiple threads per process?

- Example: Lots of shared state and blocking operations [TB15]
  - Word processor: Read input, format output, write backup file



Multiples Processes vs. Multiple Threads

- Whether to use multiple processes or threads depends on the activity [TB15]
- Processes rarely share data, and if they do, they do it explicitly.
- Closely related activities share data which favors threads.



Thread Libraries

- Thread libraries provide an API for creating and managing threads
- Pthreads
  - POSIX API for thread management and synchronization (IEEE 1003.1c)
  - API specifies behavior of the thread library ($>$ 60 API calls)
    - Internal details are up to the specific implementation of the library
  - Common in UNIX operating systems (Linux, Mac OS X, Solaris, AIX)

Basic POSIX Thread API

- Each **Pthread** is associated with
  - an identifier (Thread ID, TID)
  - a set of registers (including IP and SP)
  - a stack area to hold the execution state (functions/local vars) of that thread
- **Pthread_create** Create a new thread
  - Pass: pointer to **pthread_t** (will hold TID after successful call)
  - Pass: attributes, start function, and arguments
  - Returns: **0** on success or error value
- **Pthread_exit** Terminate the calling thread
  - Pass: exit code (casted to a void pointer)
  - Free's resources (e.g., stack)
- **Pthread_join** Wait for a specific thread to exit
  - Pass: **pthread_t** to wait for (or -1 for any thread)
  - Pass: Pointer to pointer for exit code
  - Returns: 0 on success, otherwise error value
- **Pthread_yield** Release the CPU to let another thread run

Pthread Example

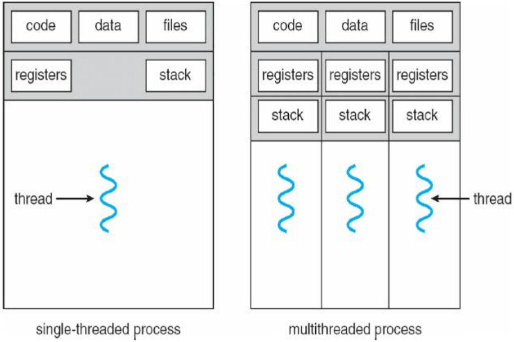
```c
void* greet( void *id )
{
   printf( "Hello, I am %ld\n", (intptr_t) id );
   pthread_exit( (void*) 0 );
}

int main()
{
   pthread_t threads[NUM];
   for( int i = 0; i < NUM; ++i )
   {
      int status = pthread_create( threads + i, NULL,
                                   greet, (void *) (intptr_t) i );
      if( status != 0 )
         die( "Error creating thread" );
   }
   for( int i = 0; i < NUM; ++i )
      pthread_join( threads[i], NULL );
   return 0;
}
```

Data Structures for Processes and Threads

- Processes group resources
- Threads encapsulate execution

→ Each of those abstractions requires different data



[SGG12]

PCB vs. TCB

- We differentiate between
  - Process Control Block (PCB): Information needed to implement processes
  - Thread Control Block (TCB): Per thread data
- Typical items in each category are:

PCB	TCB
Address space	Instruction pointer
Open files	Registers
Child processes	Stack
Pending alarms	State

- The PCB is always known to the OS
- Whether or not the OS kernel knows about threads or not depends on the thread model

Thread Model Overview

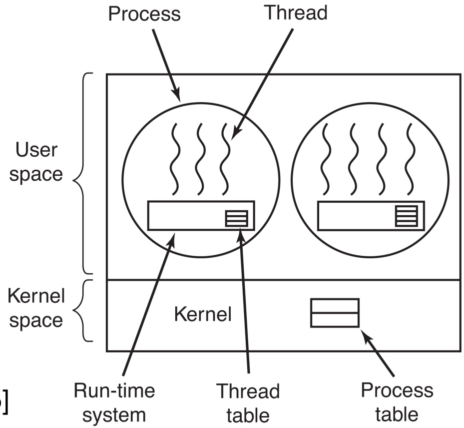
- The OS kernel always knows of at least one thread per process
  - Threads that are known to the OS kernel are called kernel threads
  - Threads that are known to the process are called user threads
- Threads can be fully implemented in user-space
  - Many-to-One Model: The kernel only knows one of possibly multiple threads
  - User threads in this model are called User Level Threads (ULT)
- The kernel can be fully aware of and responsible for managing threads
  - One-to-One Model: Each user thread maps to a kernel thread
  - User threads in this model are called Kernel Level Threads (KLT)
- The kernel can know of multiple threads per process, yet there are even more threads known to the process
  - M-to-N Model: Flexible mapping of user threads to less kernel threads
  - Also known as hybrid thread model

Many-to-One Model: User Level Threads (ULT)

- Kernel only manages process → multiple threads unknown to kernel
- Threads managed in user-space library (e.g., GNU Portable Threads)

+ Faster thread management operations (up to 100 times)
+ Flexible scheduling policy
+ Few system resources
+ Can be used even if the OS does not support threads

– No parallel execution
– Whole process blocks if only one user thread blocks
– Need to re-implement parts of the OS (e.g., scheduler)[TB15]

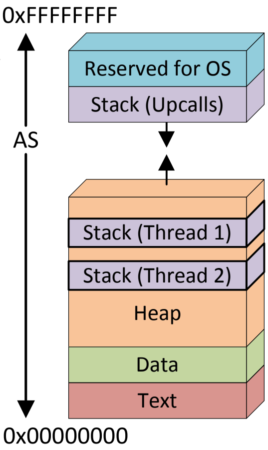


ULT Implementation

- Unix like systems (e.g., Linux) define
  - the types **`mcontext_t`** and **`ucontext_t`** to keep thread state
  - **`makecontext`** Initialize a new context
  - **`getcontext`** Store currently active context
  - **`setcontext`** Replace current context with different one
  - **`swapcontext`** User-level context switching between threads
- Using those functions, calls for creating threads, yielding, wait, can easily be implemented fully in user-space
  - e.g., yield saves own context and replaces itself with a different context
- Periodic thread switching can be implemented using a **`SIGALRM`** signal handler
- We will distribute an example how to use these function with an assignment in the tutorials (**`ult.h`**/**`ult.c`**)
- An alternative interface to the context would be **`setjmp`** and **`longjmp`**

Address-Space Layout with Two User Level Threads

- Stack
  - "Main stack" known to OS is used by thread library (e.g., called via **`SIGALRM`** upcalls)
  - Own execution state ($\equiv$ stack) for every thread is allocated dynamically by user thread library on the heap using **`malloc`**
- Heap
  - Concurrent heap use possible
  - Attention: not all heaps are reentrant!
- Data
  - Is divided into BSS, data and read-only data

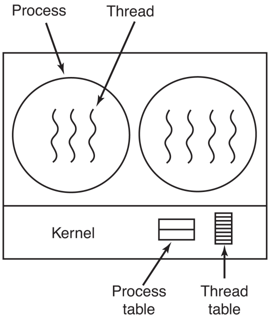


One-to-One Model: Kernel Level Threads (KLT)

- Kernel knows and manages every thread
  - Every thread known by kernel maps to one thread known by user
  - Windows, Linux, Open Solaris, Mac OS X all support this

+ Real parallelism possible
+ Threads block individually

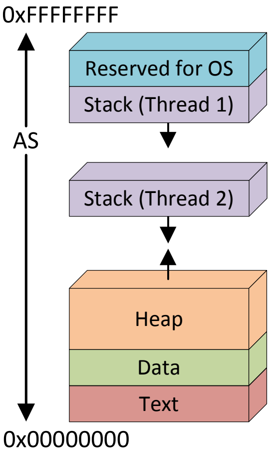
– OS manages every thread in the system (TCB, stacks, . . .)
– Syscalls needed for thread management
– Scheduling fixed in OS



[TB15]

Address-Space Layout with Two Kernel Level Threads

- Stack
  - Own execution state ($\equiv$ stack) for every thread
- Heap
  - Parallel heap use possible
  - Attention: not all heaps are thread-safe!
  - Even if the heap is thread-safe:
    Not all heap implementations perform well with many threads
- Data
  - Is divided into BSS, data and read-only data



KLT Implementation and Issues

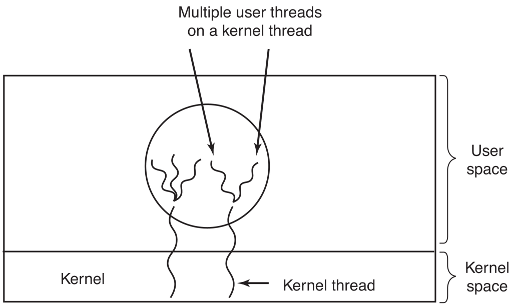
- All thread management data is stored in the kernel
- Thread management functions are provided as syscalls

- What happens when a process with multiple KLTs calls **`fork`**?

- Signals are used in UNIX systems to notify a process that a particular event has occurred
  - e.g., process can ask OS to send **`SIGALRM`** after a specific time
  - signal handler can run
    - on the process stack
    - on a stack, dedicated to the specific signal handler
    - on a stack, dedicated to all signal handlers
  - Whom is the signal delivered to?
    - All threads in process?
    - One thread that receives all signals?
    - The thread that set up the handler? What if multiple threads subscribe this signal?

M-to-N Model: Hybrid Threads

- **M** ULTs are mapped to (at most) **N** KLTs [TB15]
  - Goal: pros of ULT and KLT – non-blocking with quick management
  - Create a “sufficient” number of KLTs (kernel is only aware of KLTs)
  - Flexibly allocate ULTs on those KLTs



\+ Flexible scheduling policy

\+ Efficient execution

– Hard to debug

– Hard to implement (blocking, variable number of KLTs)

e.g., Solaris 9 and earlier, Windows NT/2000 with ThreadFiber, Linux 2.4 [MSLM91][KAR$^+$06]

Hybrid Thread Implementation: Scheduler Activations

- Goal: Don't involve kernel on thread activities such as `create` and `join`
- Idea: Map multiple ULTs on each KLT
  - When a ULT blocks (e.g., page fault, syscall) the user-space run-time system can dispatch a different ULT after being notified by the kernel
- Approach: Upcalls
  - The kernel notices that a thread will block and sends an upcall to the process
  - The upcall notifies the process of the thread id and event that happened
  - The upcall handler of the process can then schedule a different thread in that process
  - The kernel later informs the process that the blocking event has finished via another upcall

[ABLL91]

Making Single-Threaded Code Multithreaded

- It is hard to make single-threaded code multithreaded
- Not all state should be shared between threads
  - **`errno`** contains the error number of the last syscall (0 on no error)
  - **`errno`** is overwritten on subsequent system calls
  - Which thread does the current value belong to?
- Much existing code, including many libraries, are not re-entrant
  - **`malloc`** is not always thread-safe
  - **`strtok`** is not thread-safe (use **`strtok_r`**)
  - Generally: use **`_r`** variants of functions (**`rand_r`** instead of **`rand`**)
- How should stack growth be managed?
  - Normally the kernel grows the (single) stack automatically when needed
  - What if there are multiple stacks?

References I

[ABLL91] 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. *SIGOPS Oper. Syst. Rev.*, 25(5):95–109, September 1991.

[KAR+06] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, page 133–145, New York, NY, USA, 2006. Association for Computing Machinery.

[MSLM91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. *SIGOPS Oper. Syst. Rev.*, 25(5):110–121, September 1991.

References II

[RDF+20] Florian Rommel, Christian Dietrich, Daniel Friesel, Marcel Köppen, Christoph Borchert, Michael Müller, Olaf Spinczyk, and Daniel Lohmann. From global to local quiescence: Wait-free code patching of multi-threaded processes. In *14th Symposium on Operating System Design and Implementation (OSDI '20)*, pages 651–666, November 2020.

[SGG12] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 4th edition, 2015.