Karlsruhe Institute of Technology

Betriebssysteme

5. Threads

Prof. Dr.-Ing. Frank Bellosa | WT 2016/2017

KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT) – OPERATING SYSTEMS GROUP

Where we ended last lecture

- Processes
  - Program : Recipe is like Process : Cooking
  - Processes are a resource container for the OS
  - For the process it feels like it is alone: it has its own CPU and memory
  - The OS implements multiprogramming by rapidly switching processes
- In POSIX processes duplicate themselves using **`fork`**
  - This creates a process hierarchy
  - Parent processes collect the exit status of their children by **`wait`**ing for them
- Processes may block, waiting for an event to happen
  - This happens e.g., when waiting for a system call to return (e.g., I/O, wait)
  - Blocked processes are not run by the OS until the event happens
- New programs are loaded by overwriting a duplicate with a new executable file using **`exec`**
  - Parameters and an environment can be passed to initialize the program

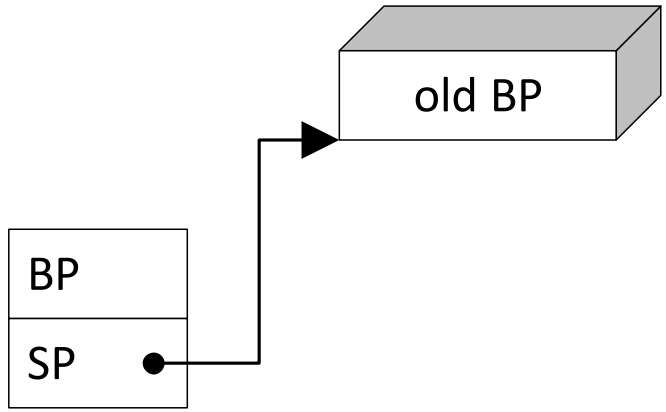
Where we ended last lecture

- The ABI standardizes the binary interface of programs and the OS
  - “Where to put stuff”
    - Process sections
    - Alignment
  - “How to talk to others” (calling conventions)
    - Within the process (functions)
    - Between processes (inter-process communication)
    - Between processes and the OS (system call)

→ Interoperability between processes, libraries, and OS

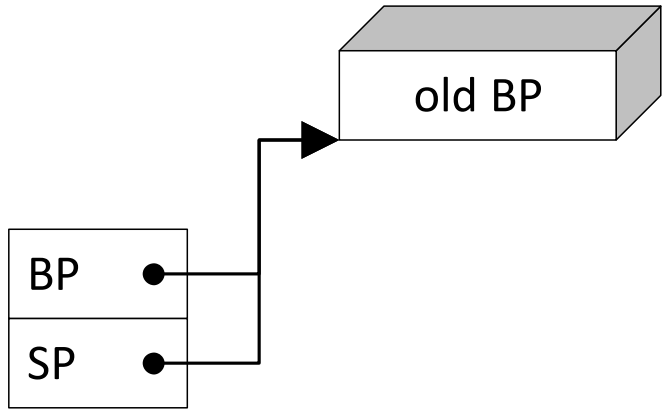
Where we ended last lecture

- The stack stores
  - Local variables
  - Execution state of a thread
- cdecl calling convention
  - Save old stack frame
  - Initialize new stack frame
  - Make room for local variables
  - Push arguments on stack in reverse order
  - Push return address and jump to function
  - Repeat with next function to call



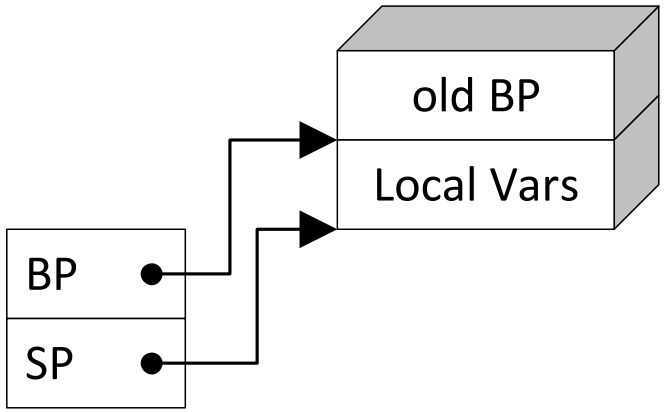
Where we ended last lecture

- The stack stores
  - Local variables
  - Execution state of a thread
- cdecl calling convention
  - Save old stack frame
  - Initialize new stack frame
  - Make room for local variables
  - Push arguments on stack in reverse order
  - Push return address and jump to function
  - Repeat with next function to call



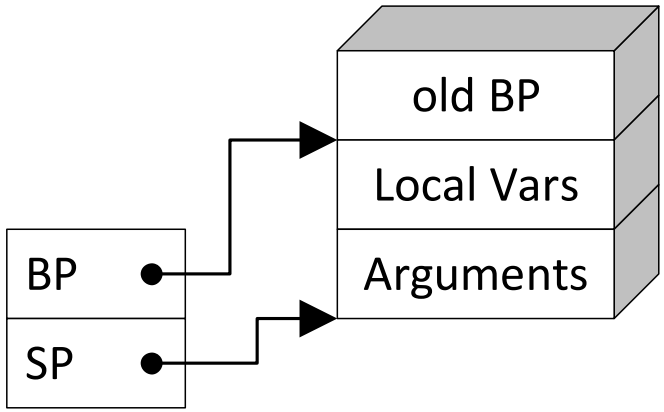
Where we ended last lecture

- The stack stores
  - Local variables
  - Execution state of a thread
- cdecl calling convention
  - Save old stack frame
  - Initialize new stack frame
  - Make room for local variables
  - Push arguments on stack in reverse order
  - Push return address and jump to function
  - Repeat with next function to call



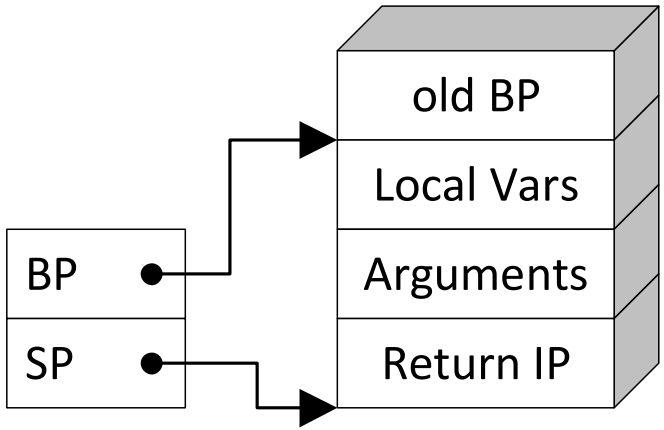
Where we ended last lecture

- The stack stores
  - Local variables
  - Execution state of a thread
- cdecl calling convention
  - Save old stack frame
  - Initialize new stack frame
  - Make room for local variables
  - Push arguments on stack in reverse order
  - Push return address and jump to function
  - Repeat with next function to call



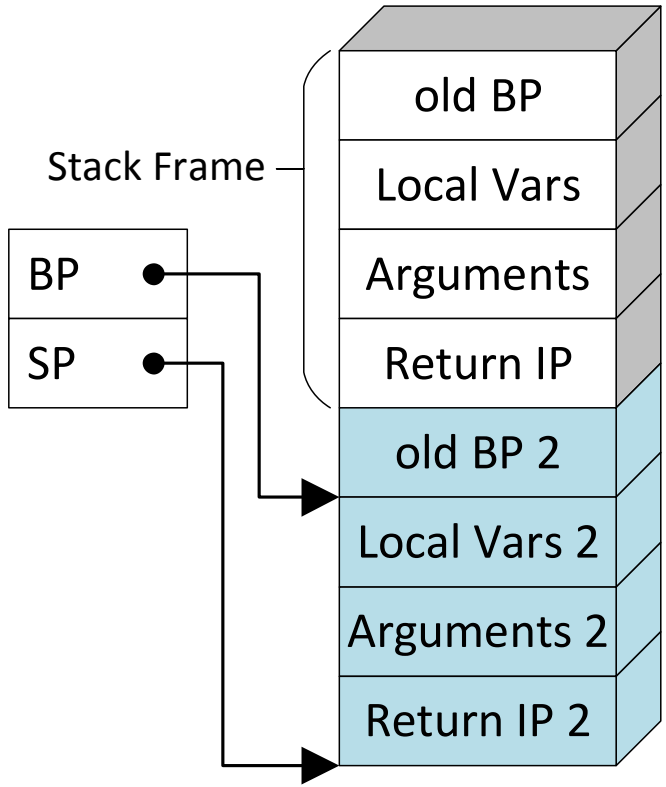
Where we ended last lecture

- The stack stores
  - Local variables
  - Execution state of a thread
- cdecl calling convention
  - Save old stack frame
  - Initialize new stack frame
  - Make room for local variables
  - Push arguments on stack in reverse order
  - Push return address and jump to function
  - Repeat with next function to call



Where we ended last lecture

- The stack stores
  - Local variables
  - Execution state of a thread
- cdecl calling convention
  - Save old stack frame
  - Initialize new stack frame
  - Make room for local variables
  - Push arguments on stack in reverse order
  - Push return address and jump to function
  - Repeat with next function to call



Threads

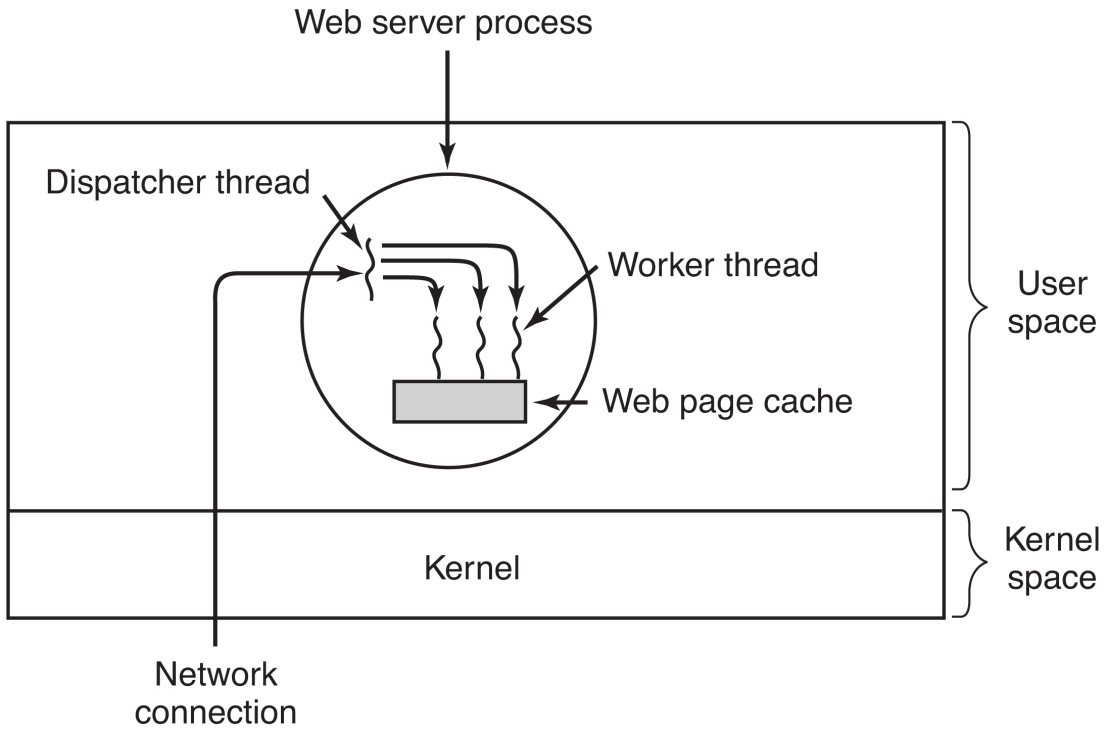
Processes vs. Threads

- In traditional OSes, each process has
  - it's own address space
  - 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/container
- Take this with a grain of salt
  - Sometimes different threads even have differing address spaces
  - In Linux threads are regular processes with shared resources and address space regions
- Bottom line:

  → Some data is thread local, some is thread global (but process local)

Why have multiple threads at all?

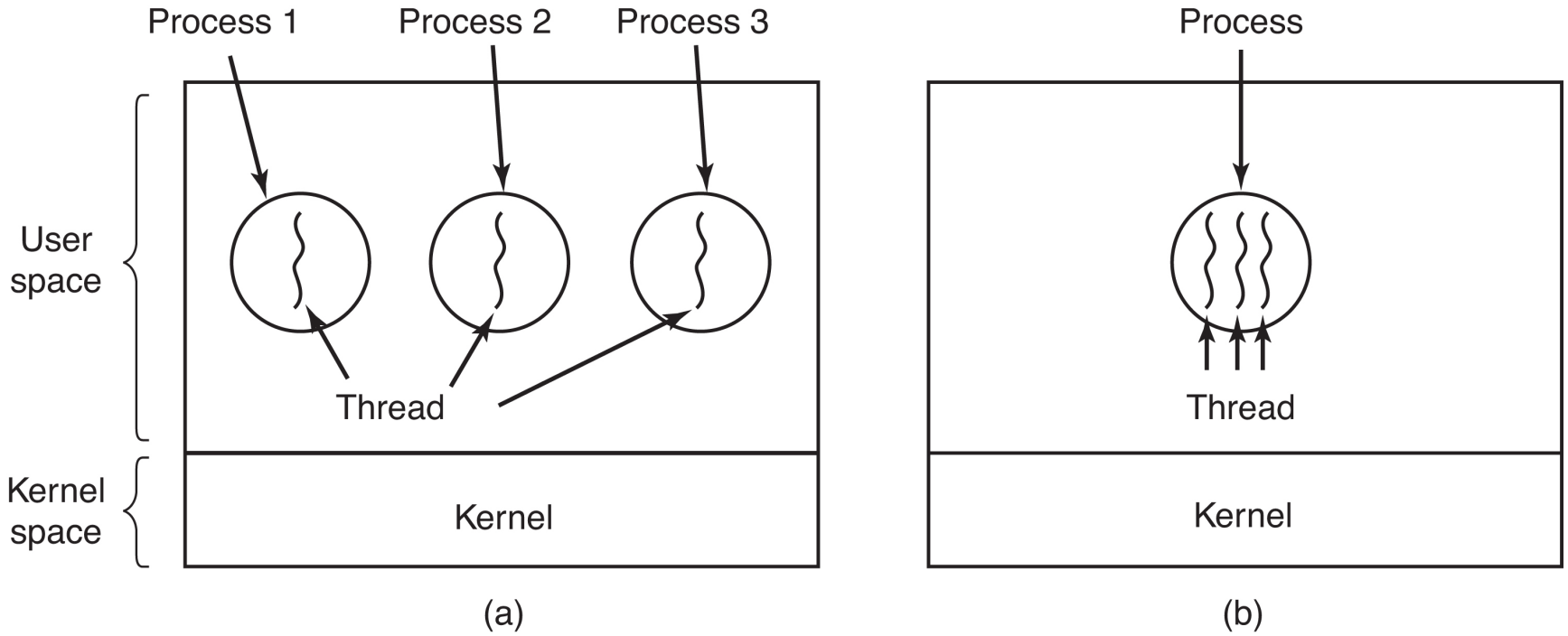
- Many programs do multiple “things” at once
- Multi-threaded web server
  - Accept new connections
  - Read request from client
  - Fetch required data
  - Process and deliver data
- Some of these activities may block



→ Writing a program as many sequential threads may be easier than dancing around blocking operations

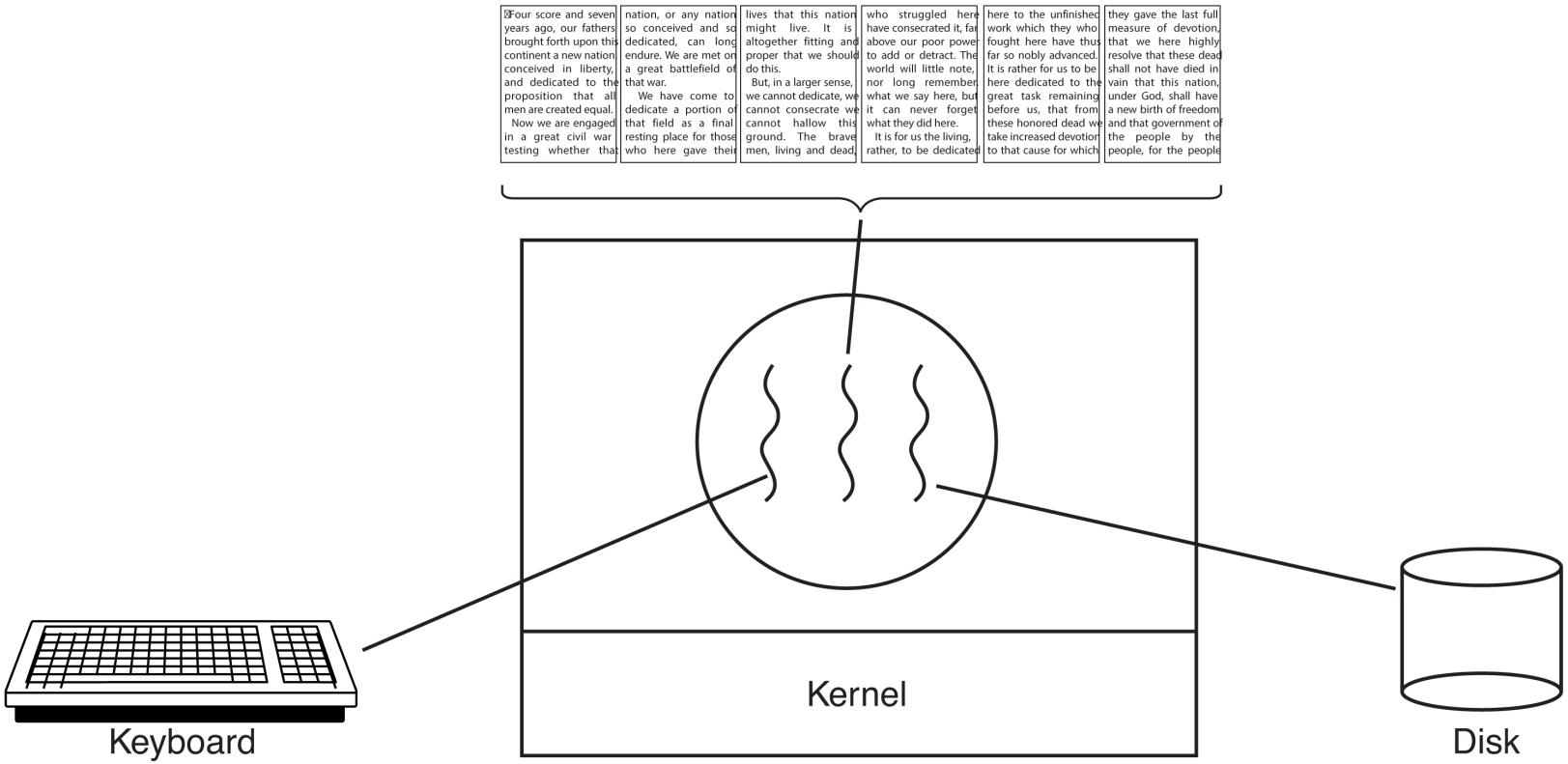
But wasn't that the reasoning behind processes?

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



Why have multiple threads per process?

- Different example with more shared state and blocking operations
  - Word processor: Read input, format output, write backup file



Thread Libraries

- Thread libraries provide an API for creating and managing threads
- Pthreads
  - POSIX API for thread creation 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 (Solaris, Linux, Mac OS X)

→ You will learn how to use pthreads in the tutorials

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

```
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;
}
```

No free lunch

- Multithreaded programming is challenging
  - Regardless: Whether multiple processes or multi-threading are used
  - Processes only share resources (e.g., memory addresses) explicitly
  - With threads there is more shared state, so more can possibly go wrong

- The programmer needs to take care of
  - Dividing, ordering, and balancing activities
  - Dividing data
  - Synchronizing access to shared data

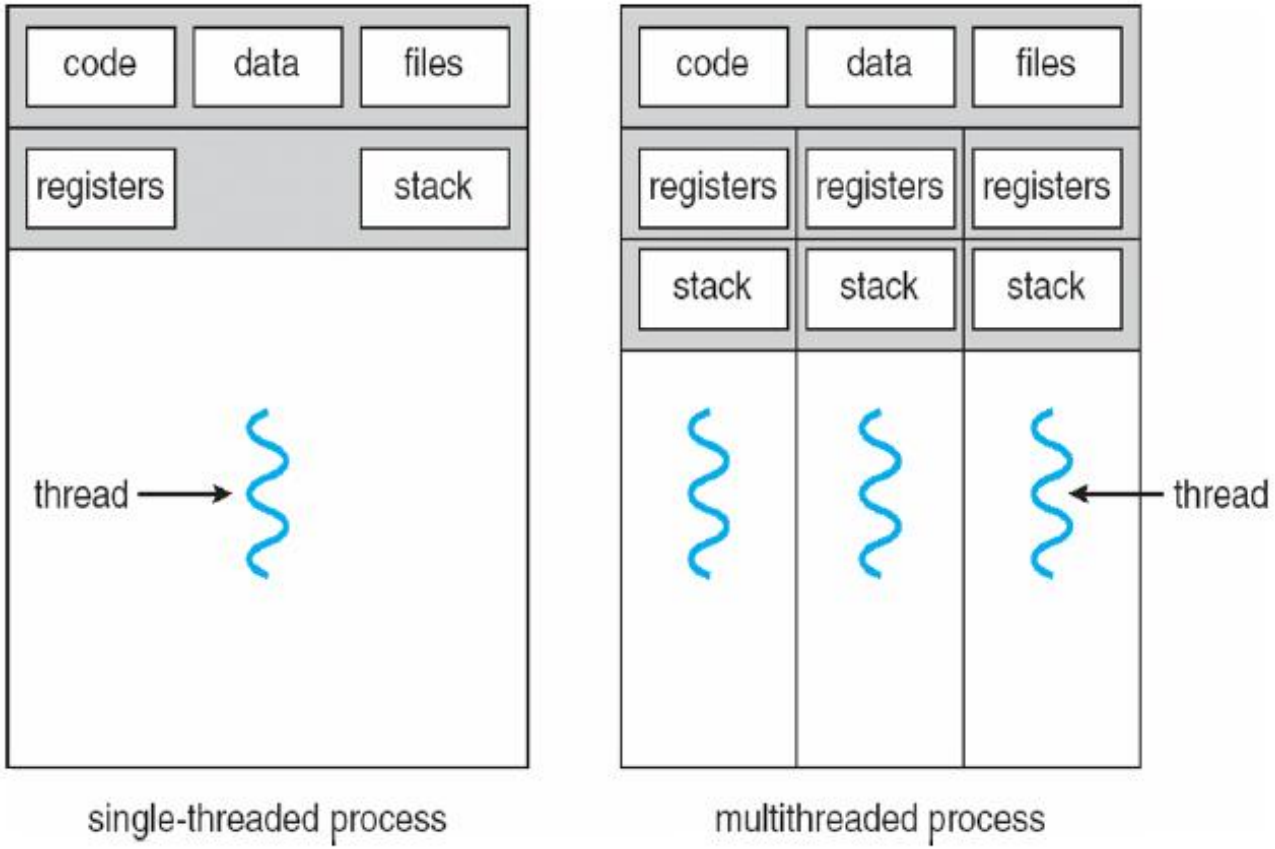
→ We will discuss synchronization in-depth in the following lectures

- Now: How threads are implemented

Process vs. Thread

- Processes group resources
- Threads encapsulate execution

→ Each of those abstractions requires different data



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
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	

- 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

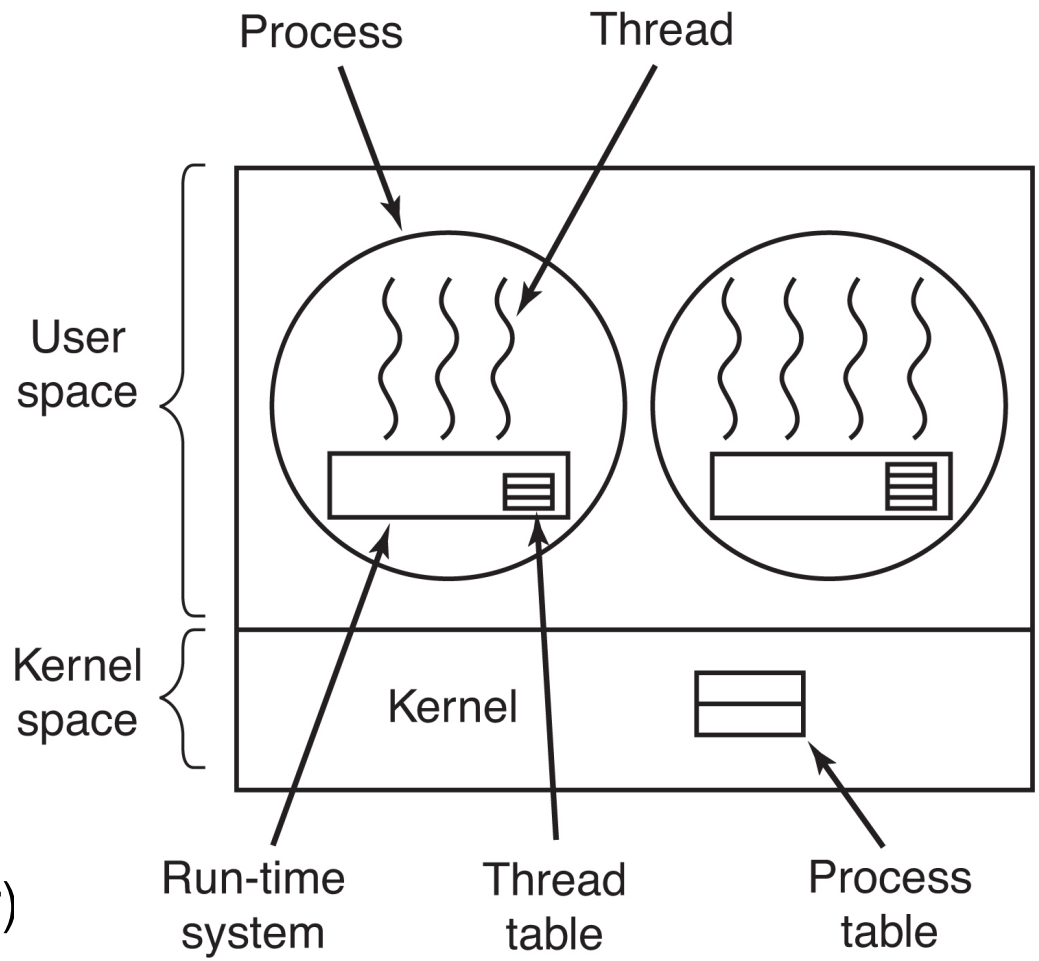
- The OS kernel always knows of at least one thread per process
  - Threads that are known to the OS kernel are called kernel threads
  - Threads that are known to the process are called user threads
- Threads can be fully implemented in user-space
  - Many-to-One Model: The kernel only knows one of 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)

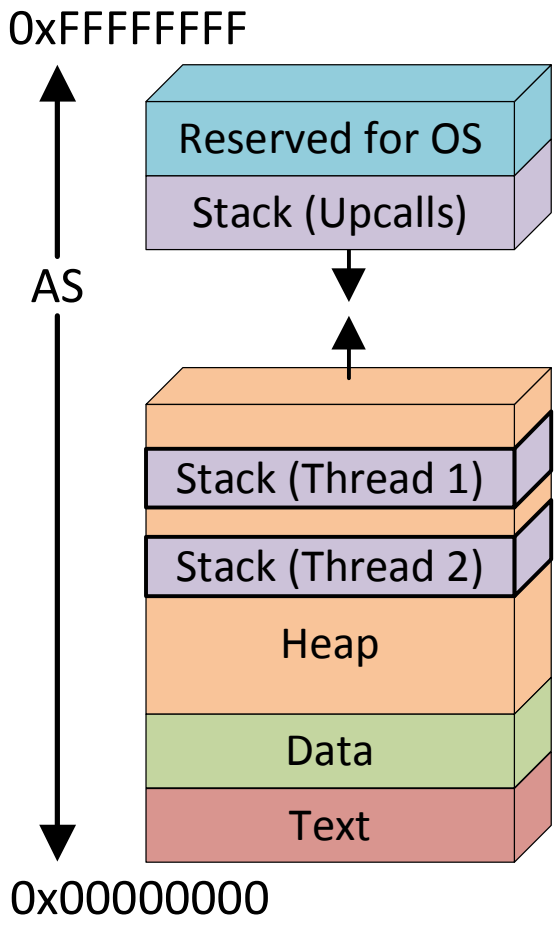


ULT Implementation

- System V 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, etc 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` exception 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`**
  - Possibly own stack for (each) exception handler
- Heap
  - Concurrent heap use possible
  - Attention: not all heaps are reentrant!
- Data
  - Is divided into BSS, data and read-only data here as well (omitted for clarity)

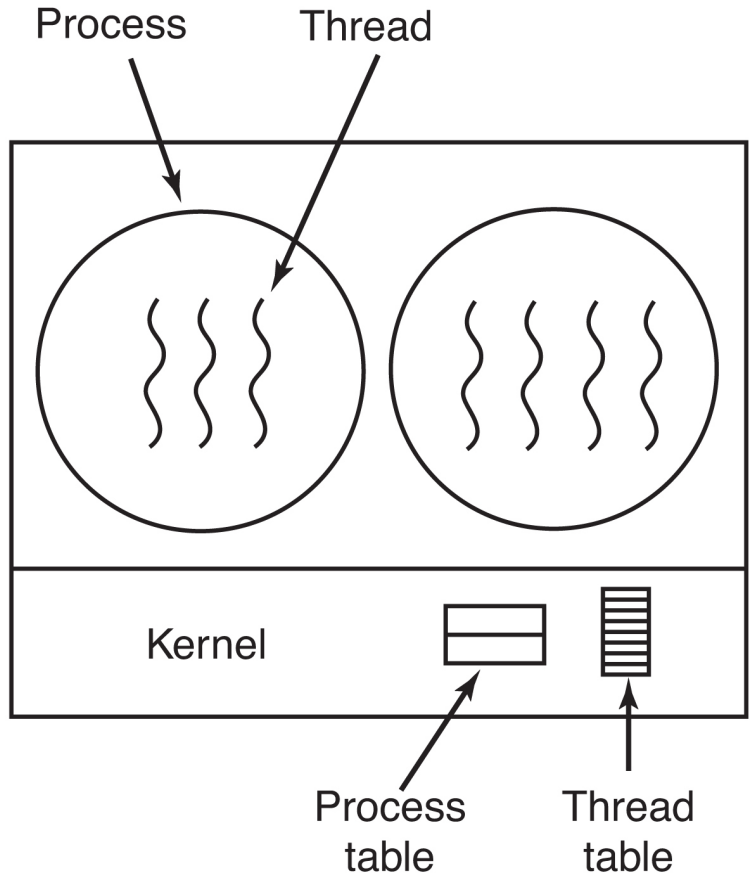


One-to-One Model: Kernel Threads (KLT)

- Kernel knows and manages every thread
  - Every thread known by kernel maps to one thread known by user
  - Windows XP/Vista/7, Linux, Solaris, Mac OS X all support this

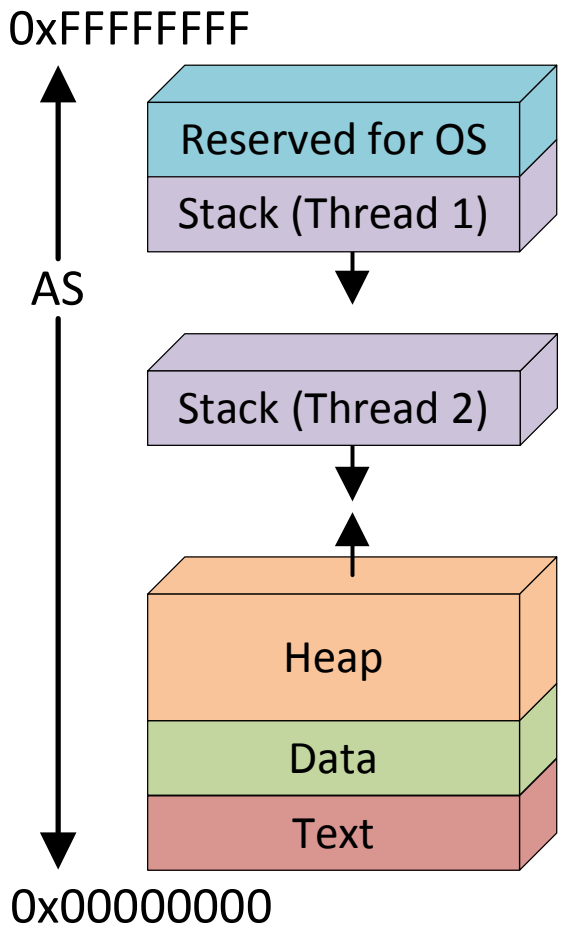
+ Real parallelism possible
+ Threads block individually

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



Address-Space Layout with Two Kernel Level Threads

- Stack
  - Own execution state ($\equiv$ stack) for every thread
  - Possibly own stack for (each) exception handler
- 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 here as well (omitted for clarity)



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
  - Who 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

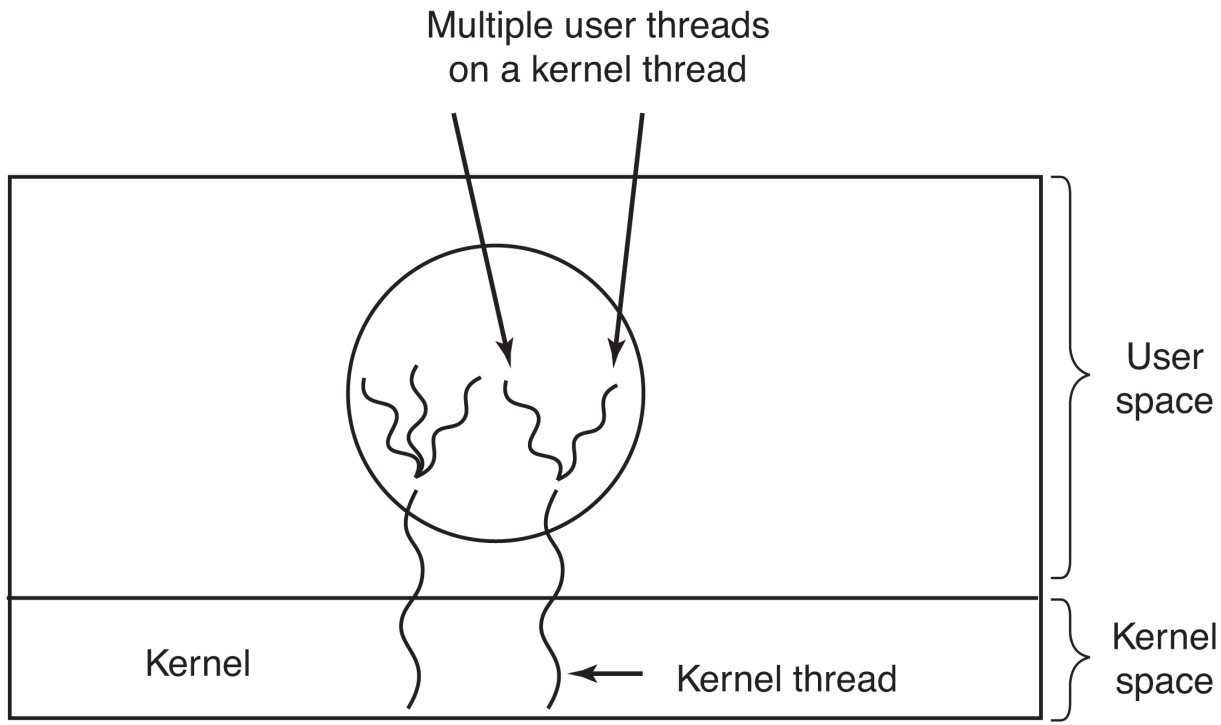
- **M** ULTs are mapped to (at most) **N** KLTs
  - Goal: pros of ULT and KLT – non-blocking with quick management
  - Create a “sufficient” number of KLTs (kernel is only aware of KLTs)
  - Flexibly allocate ULTs on those KLTs
  - e.g., Solaris 9 and earlier, Windows NT/2000 with ThreadFiber, Linux 2.4

\+ Flexible scheduling policy

\+ Efficient execution

– Hard to debug

– Hard to implement
e.g., blocking
e.g., number of KLTs



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 runs a different ULT without switching to the kernel
- Approach: Upcalls
  - The kernel notices that a thread will block and sends a signal to the process
  - This upcall notifies the process of the thread id and event that happened
  - The exception 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
- Approach is similar to calling the ULT scheduler using `SIGALRM`. It just requires a little more cooperation by the kernel.

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?

Summary

- Programs often do closely related “things” at once
- Those things can be easily mapped to the thread abstraction where multiple threads of executions operate in the same process
- We differentiate between process information (PCB) and thread information (TCB)
- There are different thread models
  - N:1 Threads are fully managed in user-space
  - 1:1 Threads are fully managed by kernel
  - M:N Threads are flexibly managed either in user-space or kernel
- Multithreaded programs operate on the same data concurrently or even in parallel
  - Accessing such data must be synchronized
  - This makes writing such programs challenging

Further Reading

- Tanenbaum/Bos, “Modern Operating Systems”, 4th Edition: Pages 85–119

- Hybrid Threads:
  - Anderson et al.: Scheduler Activations, SOSP ’91
  - Appavoo et al.: Scheduling in K42
  - Marsh et al.: First-Class User-Level Threads, SOSP ’91