Operating Systems W3L2 - Processes & Threads: Part 2

Class	Operating Systems
Created	@Sep 16, 2020 8:59 PM
Materials	03 - Processes and Threads Part 2.pdf
Reviewed	☐
Type	Lecture

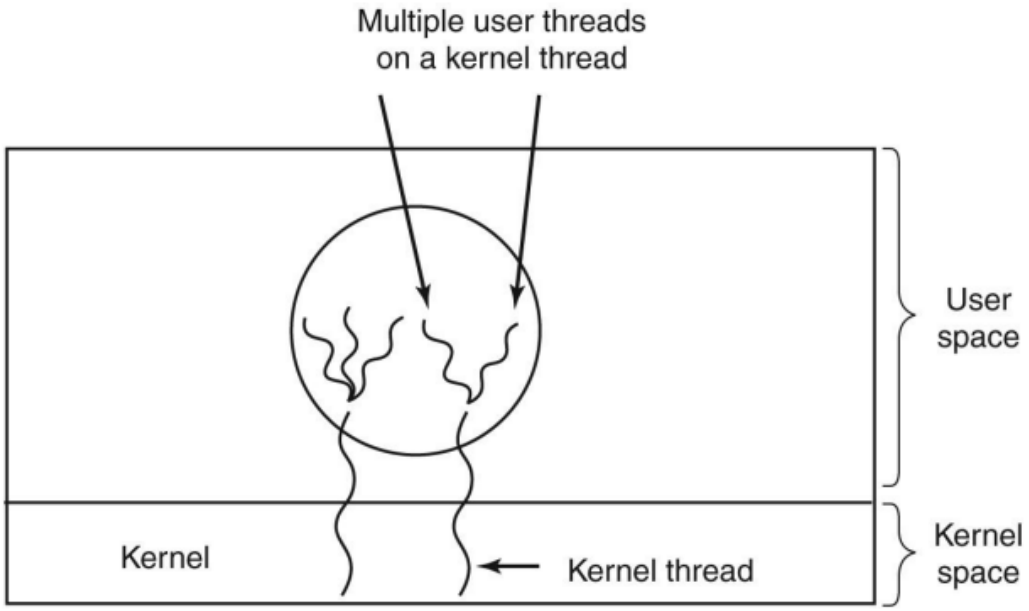
- We kicked it off with the last slides on implementing threads

Operating Systems W3L1 - Processes & Threads: Part 1 (ctd)

- Part 1 was all the theory, part 2 appears to be the down and dirty implemention
- **Kernel** level threads are still useful on a single core since the Operating System can continue execution in swapping threads

▼ Hybrid implementation

Hybrid Implementation



▼ Herbert Simon quote

Wiki page

The basic idea is that the several components in any complex system will perform particular subfunctions that contribute to the overall function.

—THE SCIENCES OF THE ARTIFICIAL,
Herbert Simon

> ! On the note of labs... keep in mind your code will need to run on linux machines / the CIMS machines

Processes vs. Threads

- **Thread:** Path of execution
  - ▼ A thread has...
    - an execution state
    - saved thread context for when it's not running
    - an execution stack
    - some per-thread static storage for local variables
    - access to the memory and resources of its process (all threads of a process **share** this)
- **Process:** Unit of resource ownership
  - The process is the unit for the *resource allocation* as well as a unit for *protection*
    - You can't give these resources to a thread (but they do have access to them)
  - A process has its own address space

▼ What is a static variable?

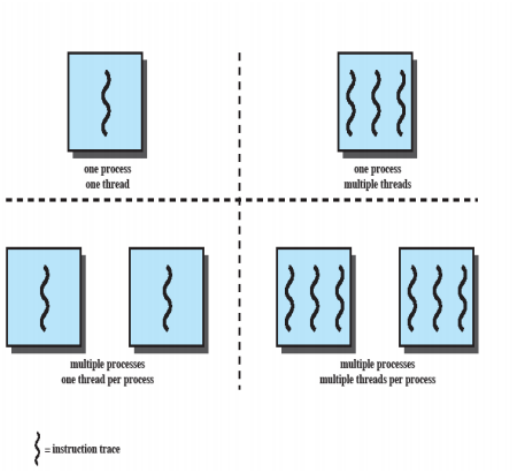
When you exit or return a function, the previous static variables will retain their values- they are stored in a different space (not the heap, which is used only for dynamic allocation)

▼ Combination of processes and threads

- Today, the bottom-right is the most common found in application. Can you think of some reasons why?

A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach ... Example: MS-DOS

A Java run-time environment is an example of a system of one process with multiple threads.



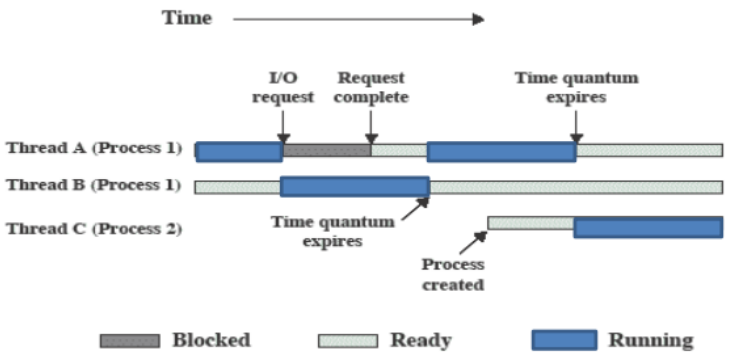
▼ What are 4 big benefits of threads?

1. Takes less time to create a thread than a process (thus you can be more volatile with them, in tandem with point 2)
2. Takes less time to terminate a thread than a process (^^)
3. Switching between two threads takes less time than switching between processes
4. Threads enhance efficiency in communication between programs

▼ Multithreading on Uniprocessor Systems

- Time Quantum ↔ Time Slice
- I/O may be working in parallel with a thread
- Time slice is *per process* (versus per thread)

Multithreading on Uniprocessor System



▼ What is a good distinction between *concurrency* and *parallelism*?

Concurrency is two threads, or processes, that are both making progress, while parallelism specifies that they must be executing at the same time

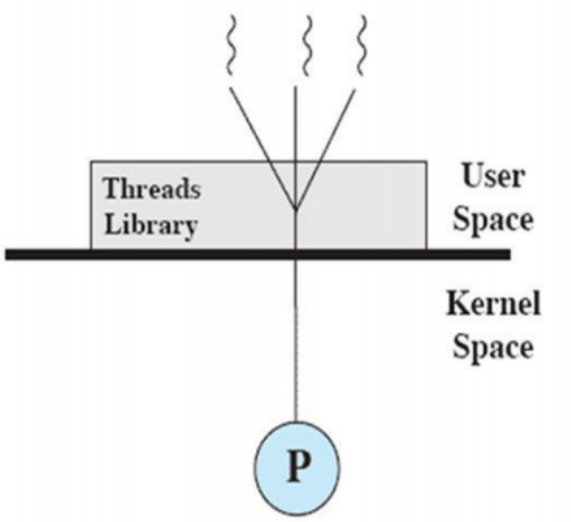
- Process and threads are scheduled differently
- We don't want CPUs to be idle to reduce latency, but also because being idle still uses some power (so we may as well use it)

User-Level vs. Kernel-Level Threads

▼ ULTs

A thread is managed entirely by application (process) and the kernel is *not* aware of its existence

- All thread management is done by the application.
- The kernel is not aware of the existence of threads.
- The kernel assigns the whole process as a single unit.



Advantages

- Thread switch does not require kernel-mode.
- Scheduling (of threads) can be application specific.
- Can run on any OS.

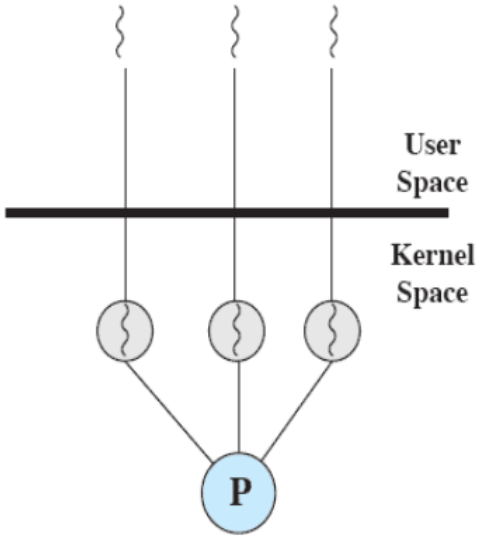
Disadvantages

- A system-call by one thread can block all threads of that process.
- In pure ULT, multithreading cannot take advantage of multiprocessing/multicore

▼ KLTs

All threads are managed in the kernel space

- Thread management is done by the kernel.
- No thread management is done by the application.
- Windows OS is an example of this approach.



Advantages

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors/cores.
- If one thread in a process is blocked, the kernel can schedule another thread of the same process.

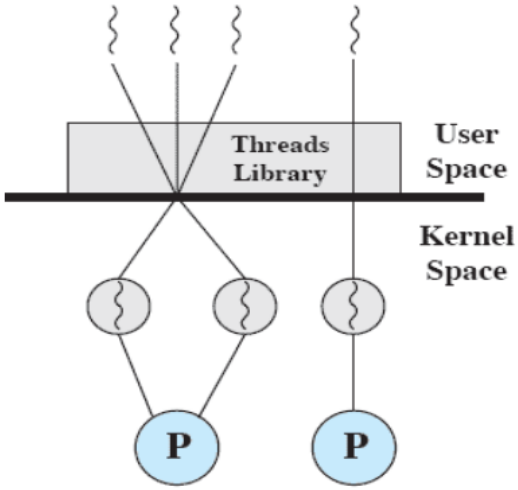
Disadvantages

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

▼ ULT / KLT Hybrid Approach

Thread creation is all in user space, then ULTs are mapped on to smaller number of KLTs, and which one gets mapped onto which ones is all dependent on the implementation chosen by the OS designer. With this mapping, we are making very good use of multiprogramming.

- Thread creation is done completely in user space.
- Bulk of scheduling and synchronization of threads is by the application (i.e. user space).
- Multiple ULTs from a single application are mapped onto (smaller or equal) number of KLTs.
- Solaris is an example.



▼ Threads and Process Relationship — 1:1, M:1, 1:N, M:N

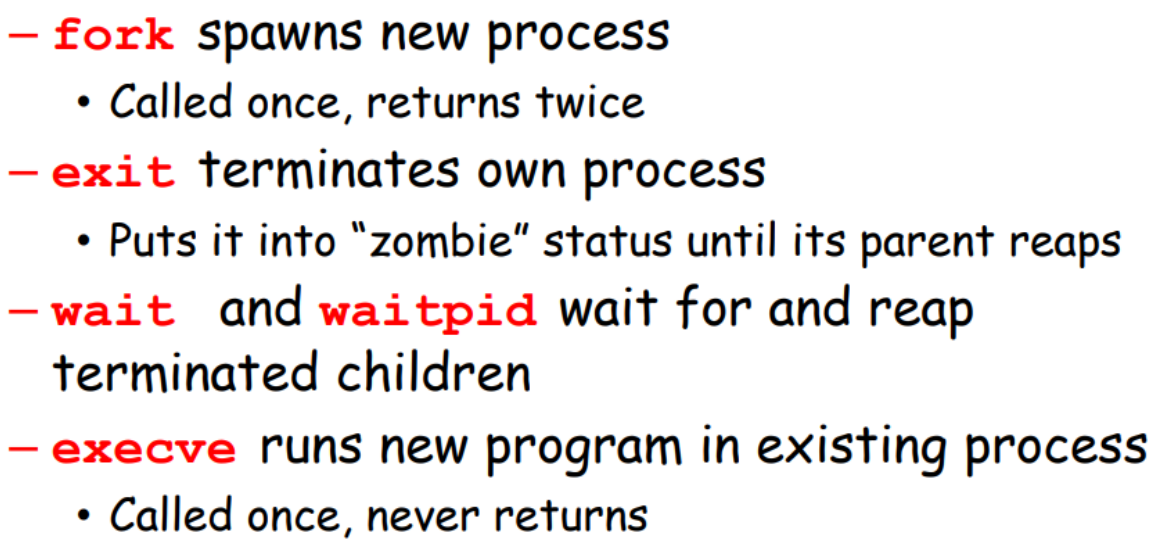
Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

APIs for Dealing With Processes (in C)

- We'll really dig into this during lab 1
  - You aren't meant to know how to do it off the bat :)

▼ Basic syscalls

`execve` is a crucial thing to know, and these calls are talked a lot about in the textbook



Fork

- Called once, but returns twice — 0 if child, the child `pid` if the parent (this is done for differentiation)
- The child process is identical to calling (parent) process in *all* aspects
  - In the case of protection/security conflicts, the `fork()` call will fail
  - *These parent/child processes do not necessarily run in parallel*
- Parent and child are still two processes → they start with the same state, but each has a private copy of memory
- Both the parent and the child can continue forking ⭐
- Forking is creating a new process (well, two- parent and child where the child is `new` and the parent continues, but they both start running from the same spot)
  - Then, running from that point → You can't ever fully determine the order of processes output

▼ Fork Example Diagram

- Both parent and child can continue forking

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```