NTNU | Norwegian University of Science and Technology

Operating Systems

Lecture 5: Threads

Michael Engel

Review: fast process creation

Unix process model

- Copying the address space takes a lot of time
  - Especially if the program immediately calls exec..() afterwards ➛ *complete waste of time!*
- Historic solution: vfork
  - The parent process is suspended until the child process calls exec..() or terminates using _exit()
- The child simply uses code and data of its parent (without copying!)
  - The child process ***must not change any data***
  - sometimes not so simple: e.g., don't call exit(), but _exit()!
- Modern solution: ***copy on write***
  - Parent and child process ***share the same code and data segments*** using the memory management unit (MMU)
  - A segment is copied only if the child process changes any data
  - This is not the case when exec..() is called directly after fork()
  - fork() using copy on write is *almost* as fast as vfork()

Can we do better?

Unix process model

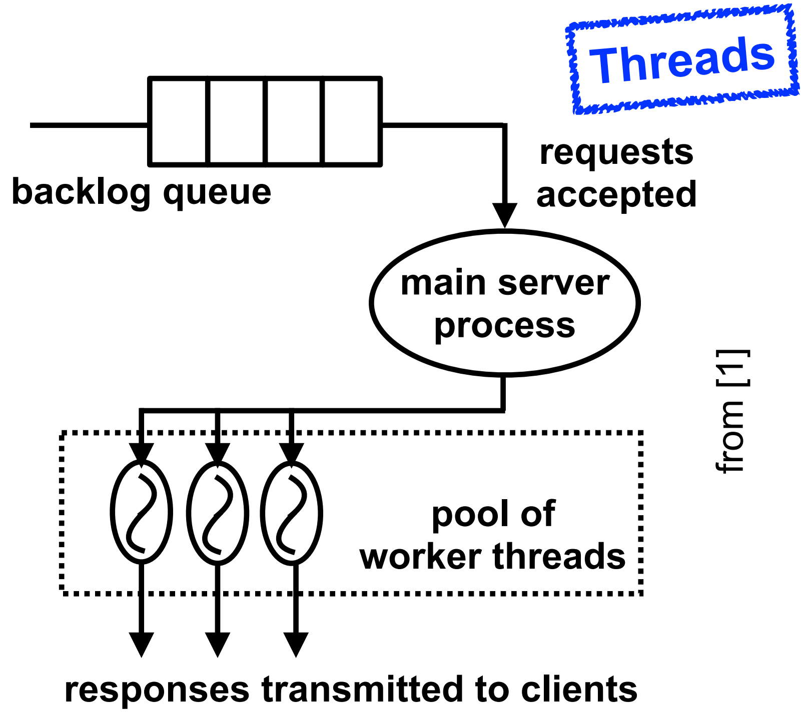
- Modern solution: ***copy on write***
  - `fork()` using copy on write is *almost* as fast as `vfork()`
- The **weight** of a process is an *informal* description of the size of its context
- Accordingly, it is an indicator for the time required for a context switch, which does (among other things):
  - CPU scheduling
  - saving the previous context
  - loading the new context
- Classical Unix processes are “heavyweight”
  - …no matter if we use copy-on-write or not

Lightweight processes (threads)

Threads

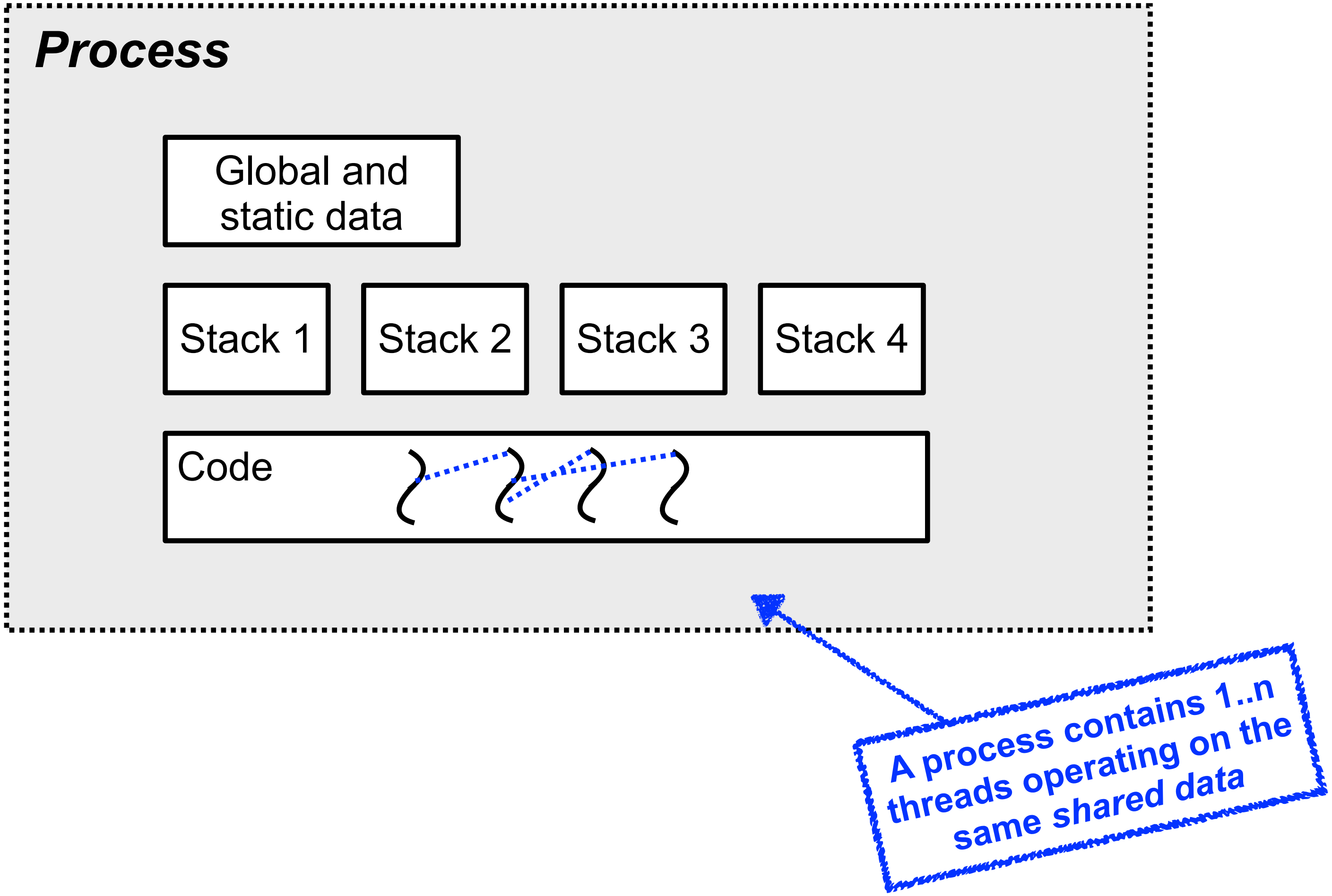
- With processes, there is a 1:1 relation between **control flow** and **address space**
  - even for forked processes due to copy-on-write
- Closely cooperating **threads** can share an address space
  - code + data + bss + heap, but **not** the stack!
  - Why not the stack?
    - Each thread has an independent flow of control
    - Accordingly, it required an independent call hierarchy, local variables etc.
- Advantage of threads:
  - Complex operations can be delegated to a lightweight helper thread
  - The parent thread can already wait for input while the helper thread is running ➛ reduced **latency** (response time)

Threads example



- Typical use case for threads: **web server**
- Programs consisting of independent control flows can immediately benefit from multiprocessor systems
- Fast context switch: no need to copy the address space
  - only switch the stack pointer – one CPU register
- Disadvantage of threads:
  - Difficult and error-prone to program
  - Access to shared data of threads requires coordination
  - OS still has to schedule threads ➛ overhead

Threads in Windows



Threads in Windows (2)

- **Process:** provides environment and address space for threads
  - But has **no execution context** in itself!
- A Win32 process always contains at least one thread
- **Thread:** unit executing code
  - Every thread has its own stack and CPU register set (especially the program counter)
  - The *scheduler* allocated compute time to the threads
- All threads are kernel level threads
  - User level threads *(fibers)* are possible, but unusual
- Strategy: Keep the number of threads low
  - Use overlapping (asynchronous) I/O

Threads in Linux

- Linux implements **POSIX threads** using the **pthreads** library
- pthreads on Linux use a Linux-specific system call:

Linux system call:

```
int __clone(int (*fn)(void*), void *stack, int flags, void *arg)
```

- Universal function, parameterized using the **flags** parameter:
  - `CLONE_VM` use a common address space
  - `CLONE_FS` share information about the file system
  - `CLONE_FILES` share file descriptors (open files)
  - `CLONE_SIGHAND` share the signal handler table

- In Linux, all **threads** and **processes** are internally managed as ***tasks***
  - The scheduler does not differentiate between those

Threads in Linux (2)

- Originally, threads of a process showed up as individual processes in the ps output [5]

```
% cc thread-pid.c -o thread-pid -lpthread
% ./thread-pid &
[1] 14608
main thread pid is 14608
child thread pid is 14610
% ps x
  PID TTY      STAT   TIME COMMAND
14042 pts/9    S      0:00 bash
14608 pts/9    R      0:01 ./thread-pid
14609 pts/9    S      0:00 ./thread-pid
14610 pts/9    R      0:01 ./thread-pid
14611 pts/9    R      0:00 ps x
```

- More recent Linux systems (from kernel 2.4) still behave like this [6], but no longer show separate processes when using CLONE_THREAD

Linux system call:

`int __clone(int (*fn)(void *), void *stack, int flags, void *arg)`

- New value for the **flags** parameter:
  - CLONE_THREAD If CLONE_THREAD is set, the child is placed in the same thread group as the calling process

Fibers

Threads

- *also called **user-level threads**, **green threads** or **featherweight processes***
- Implemented on application level only (inside of a process)
  - The operating system doesn’t know about featherweight processes
  - Accordingly, **scheduling** affects the whole process
- Implemented using a library: *user level thread package*
- Advantages:
  - Extremely fast context switch: only exchange processor registers
  - No switch to kernel mode required to switch to different fiber
  - Every application can choose the fiber library best suited for it
- Disadvantages:
  - Blocking a single fiber leads to blocking the whole process (since the OS doesn’t know about fibers)
  - No speed advantage from multiprocessor systems

Inspiration: Duff's Device

Threads

- Problem: copying 16-bit unsigned integers ("short"s) from an array into a memory-mapped output register is **slow** *(loop overhead)*:

```
send(short *to, *from, int count)
{
    do {  /* count > 0 assumed */
        *to = *from++;
    } while (--count > 0);
}
```

number of iterations reduced to 1/8th

```
send(short *to, *from, int count)
{
    register n = count / 8;
    do {
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
    } while (--n > 0);
}
```

8 copies per iteration

- **Optimization:**
  unroll the loop – execute multiple copy operations inside a single loop iteration
  ➛ reduces the loop overhead

Inspiration: Duff's Device

Threads

- Problem with loop unrolling: `count` has to be a **multiple of 8 now**!

```
send(short *to, *from, int count)
{
    register n = count / 8;
    do {
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
    } while (--n > 0);
}
```

please don't write code like this…

```
send(short *to, *from, int count)
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
    } while (--n > 0);
}
```

- Duff's solution [3]:
Introduce a **jump** into the loop body (using the C `switch` statement) to implement the first *n mod 8* iterations!

Fibers example: Protothreads

Threads

- stackless, lightweight threads, or coroutines
  - provide a blocking context cheaply using minimal memory per protothread (on the order of single bytes)
  - Developed by Adam Dunkels (SICS) [2]
  - Related approaches described in detail in [4]

The __LINE__ macro is a gcc extension to C: gives the current source code line number

```
#include "pt.h"
// … protothreads example …
PT_THREAD(example(struct pt *pt)) {
  PT_BEGIN(pt);

  while (1) {
    if (initiate_io()) {
      timer_start(&timer);
      PT_WAIT_UNTIL(pt,
        io_completed() ||
        timer_expired(&timer));
      read_data();
  }
}
```

```
// protothreads implementation: pt.h
#define PT_BEGIN(pt) \
    switch(pt->lc) { case 0:

// … more macros defined …
#define PT_WAIT_UNTIL(pt, c) \
     pt->lc = __LINE__; case __LINE__: \
     if(!(c)) return 0
```

Note: you don't need to understand the details here – it's a nice challenge for your C knowledge to expand the macros and find out what is going on

Processes vs. threads vs. fibers

	Processes	Threads	Fibers
Address space	separate	common	common
Kernel visibility	yes	yes	no
Scheduling	kernel level	kernel level	user space
Stack	separate per process	separate per thread	can be common
Switching overhead	very high	high	low

Conclusion

- Traditional Unix process creation using `fork` is too heavyweight for some applications
  - e.g. a heavily used web server
- Alternatives exist:
  - (kernel-level) threads
  - (user-level) fibers
- Each solution has its own advantages and drawbacks
  - Processes: copy and scheduling overhead
  - Threads: synchronization difficult to program
  - Fibers: no kernel management
    - blocking a fiber of a process blocks all fibers
- Linux has used the Unix process model exclusively for a long time
  - Windows (NT) didn’t have to be compatible and implemented threads from the beginning

References

1. Papastavrou, Stavros & Samaras, George & Evripidou, Paraskevas & Chrysanthis, Panos. (2003). Fine-Grained Parallelism in Dynamic Web Content Generation: The Parse and Dispatch Approach. 2888. 573-588. doi 10.1007/978-3-540-39964-3_35
2. A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems, Proc. ACM SenSys, Boulder, CO, USA, Nov 2006
3. Tom Duff, AT&T Bell Laboratories, Posting to the Usenet group comp.lang.c (August 1988): http://www.lysator.liu.se/c/duffs-device.html
4. Simon Tatham, Coroutines in C: https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html
5. M. Mitchell, J. Oldham, A. Samuel, Advanced Linux Programming, Sams 2001, ISBN 073570970X
6. U. Drepper, I. Molnar, The Native POSIX Thread Library for Linux, https://www.akkadia.org/drepper/nptl-design.pdf