Operating Systems

Tutorial 3: Threads

Named pipe

- Pipe can be opened like file and used to communicate between processes.
- Check popen(3)
- In that case the pipe is a special file on the file system.
  - One process open the file and writes
  - One process open the file and reads
- For now this is beyond scope
- This is all in Beej's guide to IPC chapter 5

Pipes and named pipes.

- Pipes create unidirectional IPC between two processes
  - One process reads from pipe
  - One process writes from pipe
  - Information is passed between processes
  - Communication is unidirectional only

Unnamed pipe

- Pipe can be created also using pipe(2) which creatd a pair of FDs consisting of a connected pipe.
- These FDs can later be dup(2)ed to other FDs.
- This can be useful for your EX2 (why?)
- Read man pipe and Beej's guide to IPC Chapter 4 for example.

Micro vs. Monolithic kernel

- Intro and we will not go deep
- OS can be big (Monolithic) or small. (micro)
- Big OS offer more services and is potentially better performing. Microkernel use user space processes to provide services. Potentially less risky (less code runs on max permission level so hopefully less bugs)
- We will not go deep only understand the terms

Outline

- **Motivation & basics**
- User-level threads
- Kernel-level threads
- Coding threads

Motivation

- Suppose editing a large WORD doc. This requires
  - User interaction: eg deleting a line and echoing it on the screen
  - Editing the whole document: eg, after a line was deleted, the rest of document is move 1 line upward
  - Automatic disk backups
- Handling multiple connections in TCP servers.
- Similar scenarios: Editing a .xls, (un)zipping, …
- One process
  - ☹ Difficult to code and does not use system resources well. CPU can work on other tasks while waiting for disk responses
- Many processes:
  - ☹ Problem: processes require shared memory API to access the same place (file) in memory
  - ☹ Long *context switches*
  - ☹ High system cost (entries in the process table)
- ☺ Solution: thread – a refined division of tasks

Threads

- When we run on code we have PC for current instruction and also stack.
- Just another PC that executes on the same processes space.
- All Variables are shared!
- Thread has a stack. But the variable on the stack are also shared.

Threads for performance

- Assumption – one CPU/Core. System is ONLY cpu bound
- Single process/single threaded application will be most efficient
- Real world – multiple CPU/cores. System has I/O (and multiple spindles), network and other wait times.
- Multi programming allows things to run in parallel (one thread can use CPU while others wait for network and file I/O)

Threads - General

- One / multiple thread(s) within a process
  - A Process always has the “main” thread.
- Allows multiple independent executions under the same process
- Possible states:
  - Running
  - Ready
  - Blocked
  - (Terminated)

Threads - Advantages

- ☺ **Share** open files, data structures, global variables, child processes, etc.
- ☺ Peer threads can **communicate** without using system calls.
- ☺ Threads are 10-100 times **faster** to create / terminate / switch than processes. On monolithic kernel systems.

Threads - Disadvantages

- ☹ Security & stability: open files, data structures, global variables, child processes etc are shared
- ☹ Signaling a thread affects all threads of that process (Slightly inaccurate. See below)
- ☹ If one thread terminates (e.g. divides by zero) all terminates

Threads vs. Processes

Processes	Threads
unique data	shared data
unique code	shared code
unique open I/O	shared open I/O
unique signal table*	shared signal table*
unique stack	unique stack (Thread can point to data on another thread stack)
unique PC	unique PC
unique registers	unique registers
unique state	unique state
heavy context switch	light context switch

* Signal handlers must be shared among all threads of a multithreaded app. However, each thread must have its own mask of pending / blocked signals:
use pthread_mask rather then sigprocmask.

Are processes dead?

- One task may not trust another task and doesn’t wont to crash if it does something wrong

e.g.

  - Debugger debugs suspicious process
  - Browser plug-ins (If something is done wrong by the plug-in do not crash the browser
- Security – we wish to create separation between vulnerable processes and not vulnerable process
  - Separate DB and Web server process despite sharing many information
- The business reasons - Separate groups
  - Chess engines (Stockfish, Leela chess zero) and developed by different groups then Chess U/I (xboard, cutechess) so processes and not threads are used.

However….

Previous slides reasons not withstanding multi-threads are now much (like 100 times) more common than multi-procees in modern systems.

But process are still used (even on modern systems) when appropriate!

Implementation dilemmas

- fork()
  - Only calling thread is duplicated per POSIX standard
- exec()
  - Does the command replace the entire process?
    - Yes.
- Divide process into threads by…
  - the user / the kernel?

Outline

- Motivation & basics
- User-level threads
- Kernel-level threads
- **Coding threads**

POSIX Threads & global variables

- Historically, POSIX functions assumed a single thread per process.
  - E.g., consider a naive implementation of `errno` in a multi threaded environment.
    - Errno is implemented as a macro calling a function
    - Today One Errno per thread.
  - Consider the implementation of malloc (See Doug Lea malloc)
- Hence, the need for reentrant functions.
- While this is supported by many standard functions, **`Linux : Gcc –pthread (copmiler and linker flag switches standard library)`**
  - **`OSX : Always in multithread mode. (cheaper because based on microkernel)`**

Threads in POSIX: pthreads

int **pthread_create**
(pthread_t* thread, pthread_attr_t* attr, void* (*start_func)(void*) , void* arg)

Creates a new thread of control that executes concurrently with the calling thread.

On success, the identifier of the newly created thread is stored in the location pointed by the *thread* argument, and a 0 is returned. On error, a non-zero error code is returned.

attr enables applying attributes to the new thread (e.g. detached, scheduling-policy). Can be NULL (default attributes).

start_func is a pointer to the function the thread will start executing. The function receives one argument of type *void* and returns a void**.

arg is the parameter to be given to *func*.

pthread_t **pthread_self** ()

Returns this thread's identifier.

Threads in POSIX (pthreads) – cont.

int **pthread_join** (pthread_t th, void** thread_return)
Suspends the execution of the calling thread until the thread identified by *th* terminates.
On success, the return value of *th* is stored in the location pointed by *thread_return*, and a 0 is returned. On error, a non-zero error code is returned.
At most one thread can wait for the termination of a given thread. Calling **pthread_join** on a thread *th* on which another thread is already waiting for termination returns an error.
th is the identifier of the thread that needs to be waited for
thread_return is a pointer to the returned value of the *th* thread (can be NULL).

void **pthread_exit** (void* ret_val)
Terminates the execution of the calling thread. Doesn't terminate the whole process if called from the main function.
If *ret_val* is not null, then *ret_val* is saved, and its value is given to the thread which performed *join* on this thread; that is, it will be written to the *thread_return* parameter in the **pthread_join** call.

Threads in other systems

- Threads are identical concept (different API) in all systems
- CreateThread in MSDN (Windows)
- NSThread in Cocoa (Apple)
- java.lang.Thread (Java)
- System.Threading (C#)
- THREAD (Cobol, Z/OS)

Multiprogramming design patterns

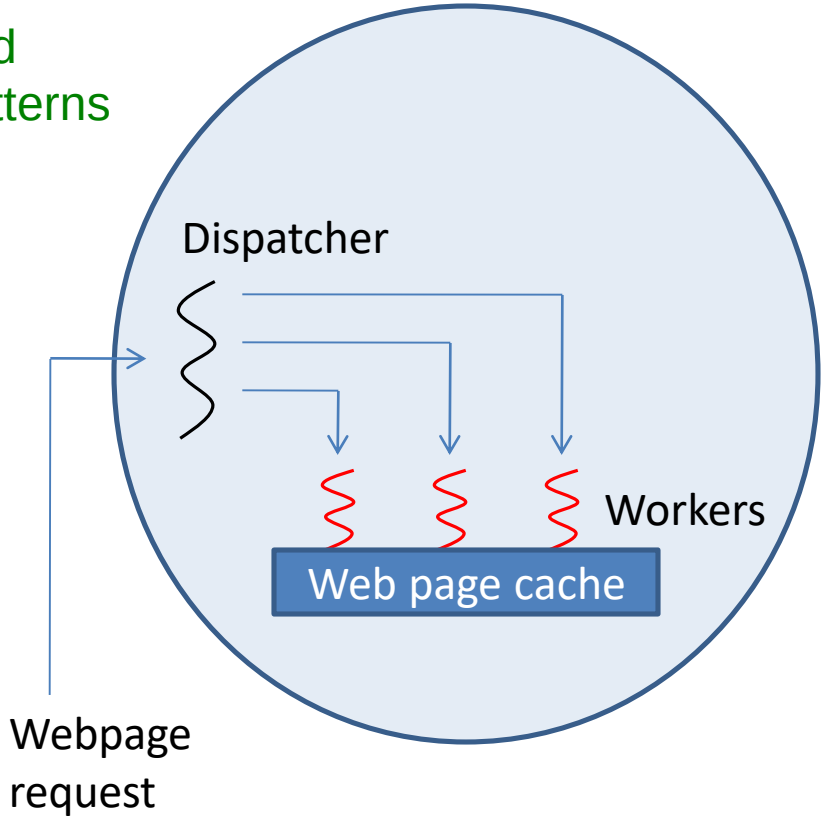
- Active Object
  - Thread has queue. Receive requests (jobs) and handle them
- Pipeline
  - Multiple active objects each doing some work and feeding the next
  - Common : streaming (one AO to read file, one AO to decode Video, one AO to decode audio, One AO for resize, One AO for subtitles, two AO for audio/video playback)

Threads - Motivation

```
Dispatcher thread:
    while (TRUE) {
        get_next_request(&buf);
        handoff_work(&buf);
    }

Worker thread:
    while (TRUE) {
        wait_for_work(&buf);
        look_for_page_in_cache(&buf, &page);
        if (page_not_in_cache(&page))
            read_page_from_disk(&buf, &page);
        return page(&page);
    }
```

See thread design patterns



Why are threads better in this case?

Example from "Modern Operating Systems", 2nd Edition, pg. 88

Singleton

- **What is Singleton pattern?**
- **Why not just use global variable?**
- **Widget * singleton == NULL**
- **Widget * instance()**
- **{**

```
If (singleton != NULL) return singleton;
else singleton=newWidget();
Return singleton;
}
```

- **Singleton == Thread safe global variable**