Threads

Daniel Hagimont (INPT)

hagimont@enseeiht.fr

http://hagimont.perso.enseeiht.fr

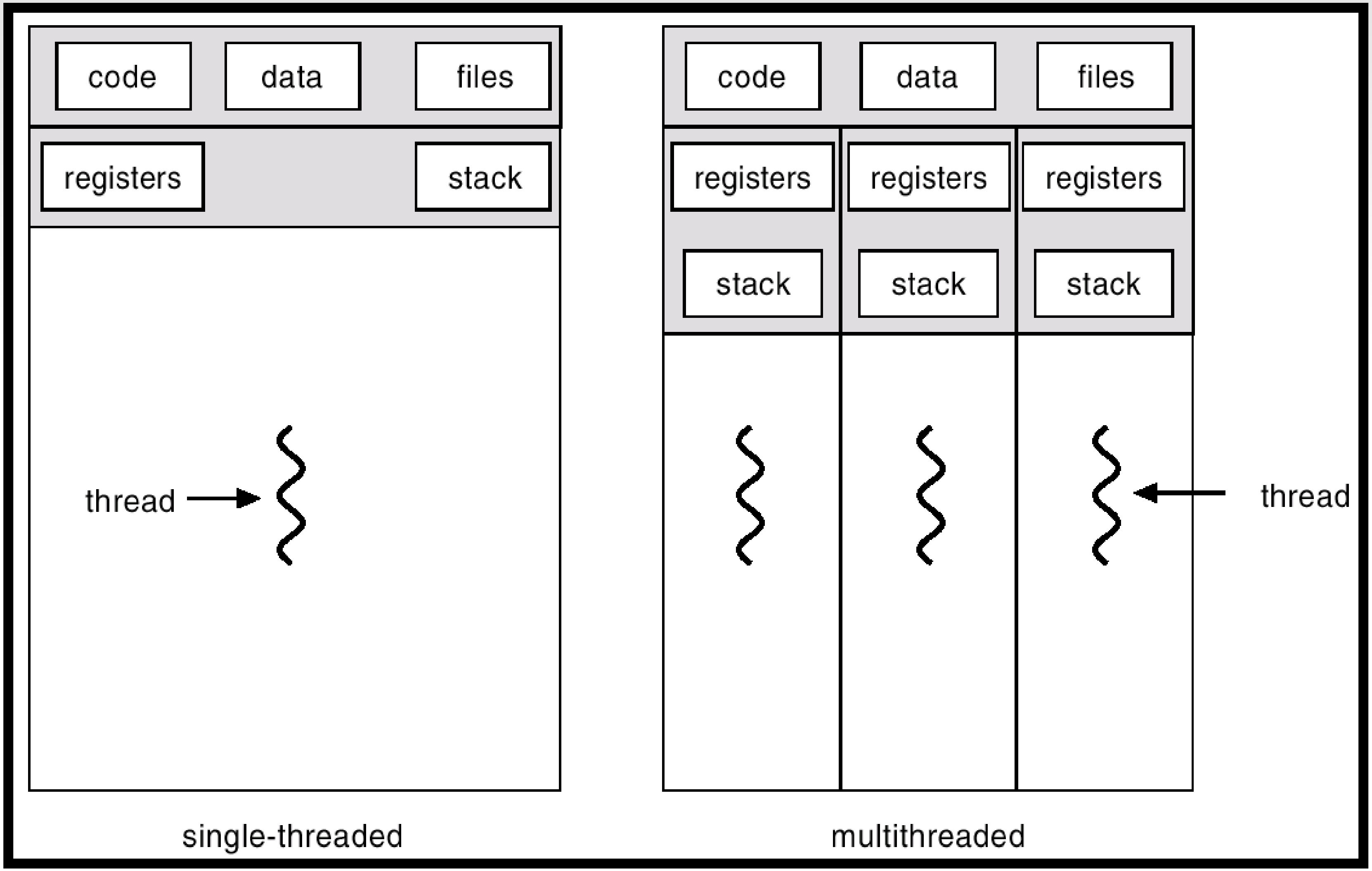
Process

- Unix process: heavy
  - Context: large data structure (includes an address space)
  - Protected address space
    - Address space not accessible from another process
    - Sharing / communication
      - At creation time (fork)
      - Via shared memory segments
      - Via messages (queues, sockets)
  - Communication is costly

Threads

- Light weight process
  - Light weight context
    - A shared context: address space, open files …
    - A private context: stack, registers …
- Faster communication within the same address space
  - Message exchange, shared memeory, synchronization
- Useful for concurrent/parallel applications
  - Easier
  - More efficient
  - Multi-core processors

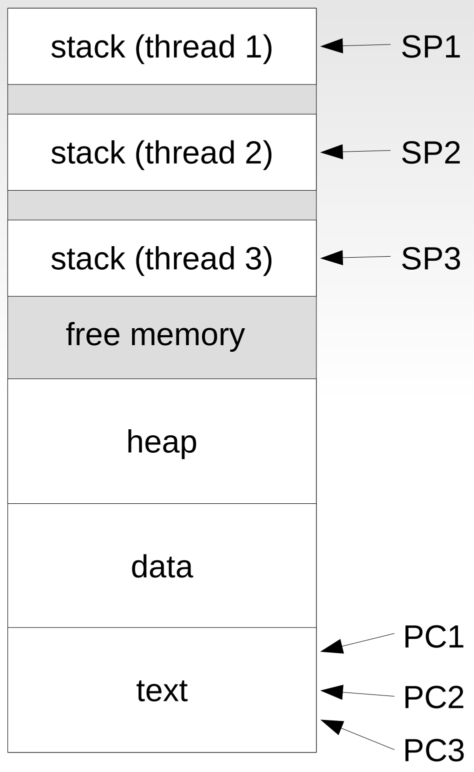
Single-threaded vs multi-threaded processes



A.Sylberschatz

Multi-threaded process

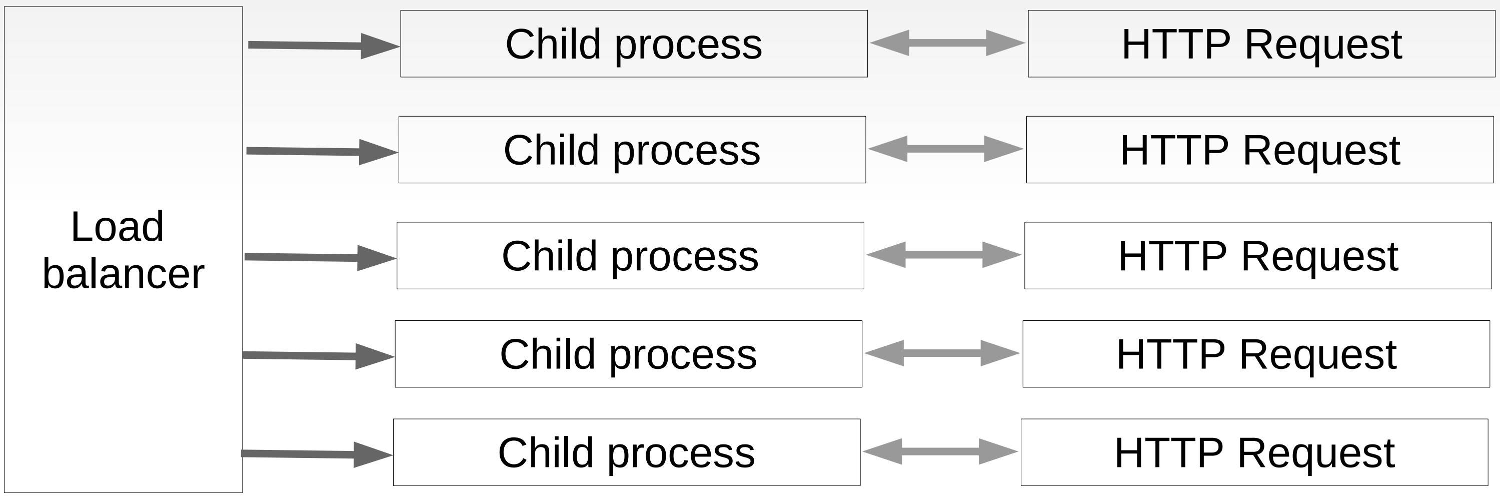
- Each thread has:
  - Private stack
  - Private stack pointer
  - Private program counter private register values
- Share:
  - Common text section (code)
  - Common data section (global data)
  - Common heap (dynamic allocations)
  - File descriptors (opened files)
  - Signals



Multi-threaded process

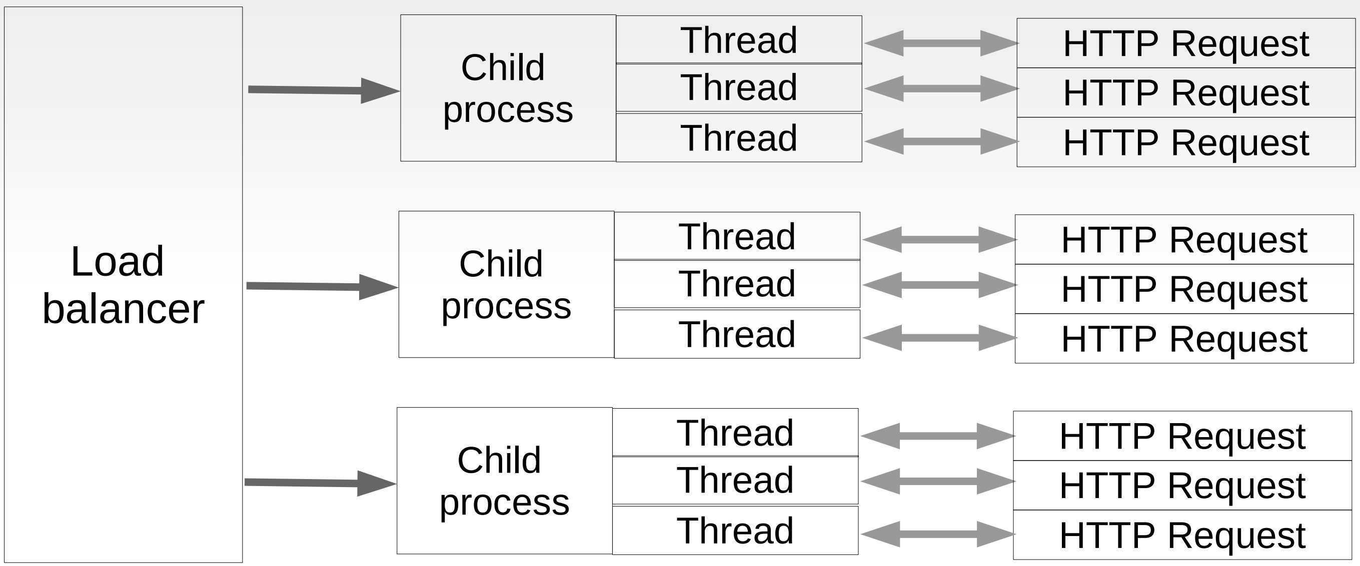
- Threads: same goal as processes
  - Do several thing at the same time
  - Increase CPU utilization
  - Increase responsiveness
- What's the difference
  - Multi-process with fork(): resource cloning
  - Multi-thread process: resource sharing

Some multi-process architectures



Apache HTTPD Prefork Model

Some multi-process architectures



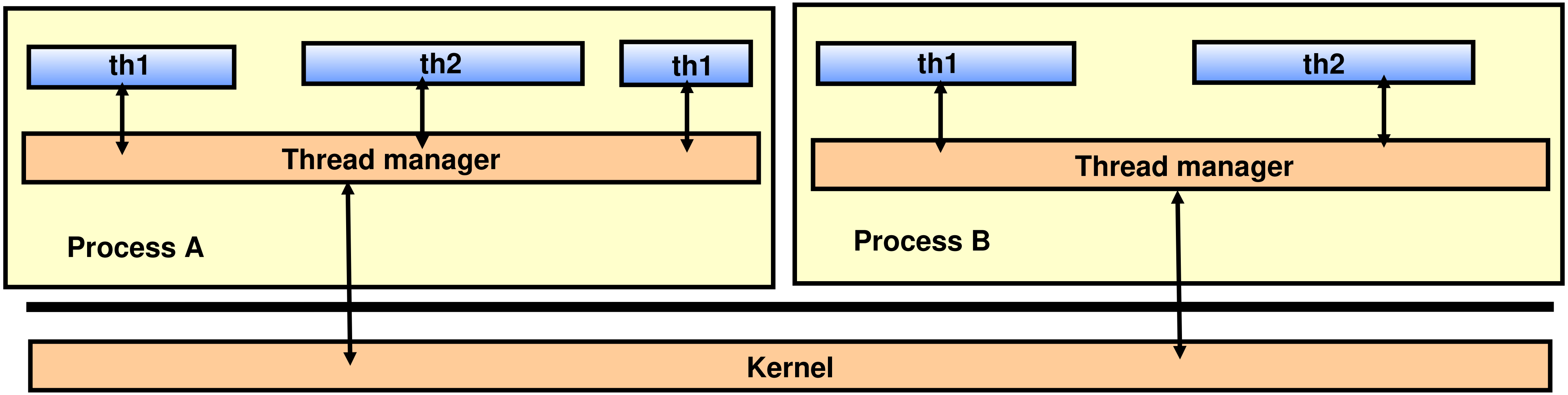
Apache HTTPD Worker Model

Exercise (thread)

- Show the number of threads for process firefox or google-chrome
  - ps with NLWP (number of lightweight processes) option
    - e.g. ps -o nlwp <processId>
  - Count number of subdirectories in /proc/<processId>/task

User-level Threads

- Implemented in a user level library
- Unmodified Kernel
- Threads and thread scheduler run in the same user process

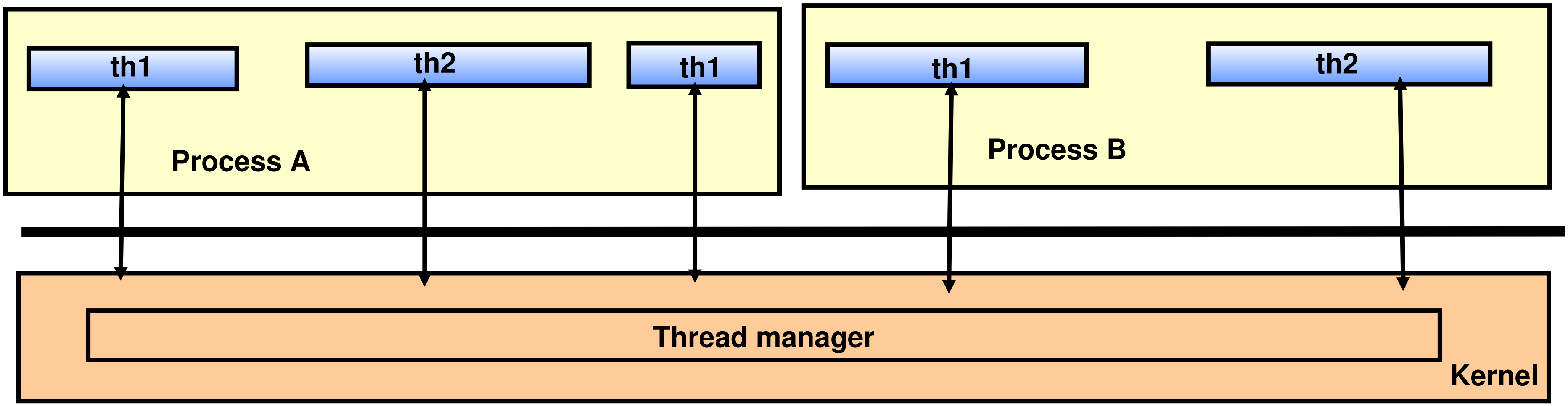


Advantages and disadvantages of User-level threads

- Parallelism (-)
  - No real parallelism between the threads within a process
- Efficiency (+)
  - Quick context switch
- Blocking system call (-)
  - The process is blocked in the kernel
  - All thread are blocked until the system call (I/O) is terminated

Kernel level threads

- Thread managed by the kernel
- Thread creation as a system call
- When a thread is blocked, the processor is allocated to another thread by the kernel



Advantages and disadvantages of Kernel-level threads

- Blocking system call (+)
  - When a thread is blocked due to an SVC call, threads in the same process are not
- Real Parallelism (+)
  - N threads in the same process can run on K processors (multi-core)
- Efficiency (-)
  - More expensive context switch / user level threads
  - Every management operation goes throught the kernel
  - Require more memory

POSIX Threads : pthreads API

- int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine)(void *), void *arg);
    - Creates a thread
- pthread_t pthread_self (void);
    - Returns id of the current thread
- int pthread_equal (pthread_t thr1, pthread_t thr2);
    - Compare 2 thread ids
- void pthread_exit (void *status);
    - Terminates the current thread
- int pthread_join (pthread_t thr, void **status);
    - Waits for completion of a thread
- int pthread_yield(void);
    - Relinquish the processor
- Plus lots of support for synchronization [next lecture]

Exercise (thread) (1/2)

```
#include <pthread.h>

void * ALL_IS_OK = (void *)123456789L;

char *mess[2] = { "thread1", "thread2" };

void * writer(void * arg)
{
  int i, j;

  for(i=0;i<10;i++) {
    printf("Hi %s! (I'm %lx)\n", (char *) arg, pthread_self());
    j = 800000; while(j!=0) j--;
  }

  return ALL_IS_OK;
}
```

Exercise (thread) (2/2)

```
int main(void)
{ void * status;
  pthread_t writer1_pid, writer2_pid;

   pthread_create(&writer1_pid, NULL, writer, (void *)mess[1]);
   pthread_create(&writer2_pid, NULL, writer, (void *)mess[0]);

   pthread_join(writer1_pid, &status);
   if(status == ALL_IS_OK)
      printf("Thread %lx completed ok.\n", writer1_pid);

   pthread_join(writer2_pid, &status);
   if(status == ALL_IS_OK)
      printf("Thread %lx completed ok.\n", writer2_pid);

   return 0;
}
```

Fork(), exec()

- What happens if one thread of a program calls fork()?
  - Does the new process duplicate all threads ? Or is the newprocess single-threaded ?
  - Some UNIX systems have chosen to have two versions of fork()
- What happens if one thread of a program calls exec()?
  - Generally, the new program replace the entire process, including all threads.

Resources you can read

- Pthreads
  - https://computing.llnl.gov/tutorials/pthreads/
- Operating System Concepts, 10th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne
  - http://os-book.com/
  - Chapters 4
- Modern Operating Systems, Andrew Tanenbaum
  - http://www.cs.vu.nl/~ast/books/mos2/
  - Chapter 2 (2.2)