Παράλληλη Επεξεργασία

Εαρινό Εξάμηνο 2023-24
«Εισαγωγή στον Πολυνηματισμό»

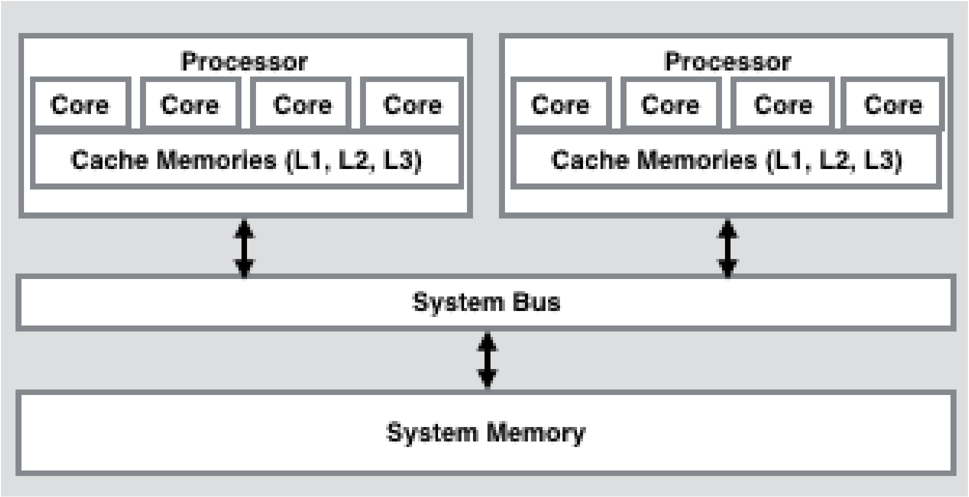
Παναγιώτης Χατζηδούκας, Ευστράτιος Γαλλόπουλος

Outline

- Processes and Threads
- POSIX Threads API
  - Thread management
  - Synchronization with mutexes
- Deadlock and thread safety

Terminology

- Parallelism in Hardware:
  - multiple cores and memory
- Parallelism in Software:
  - **process**: execution sequence within the OS, a running program
  - **thread**: can execution sequence within a process, all threads of the same process share the application data (memory)



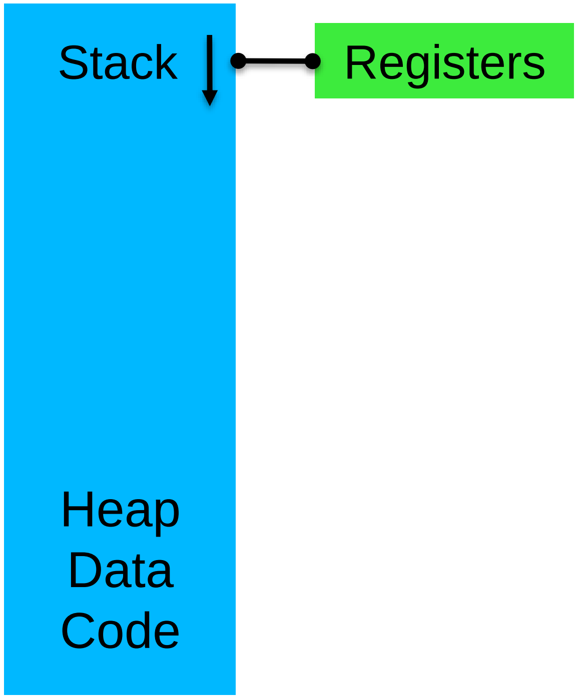

```
int a[1000];

int main( int argc, char** argv )
{
  for(int i =   0; i <  500; i++ ) a[i] = 1;
  for(int i = 500; i < 1000; i++ ) a[i] = 2;

  return 0;
}
```

Processes

- A process consists of the following:
  - Address space: text segment (code), data segment, heap and stack
  - Information maintained by the operating system (process state, priority, resources, statistics)
- Process state: snapshot where the above information has specific values
  - Memory state: state of the address space
  - Processor state: register values



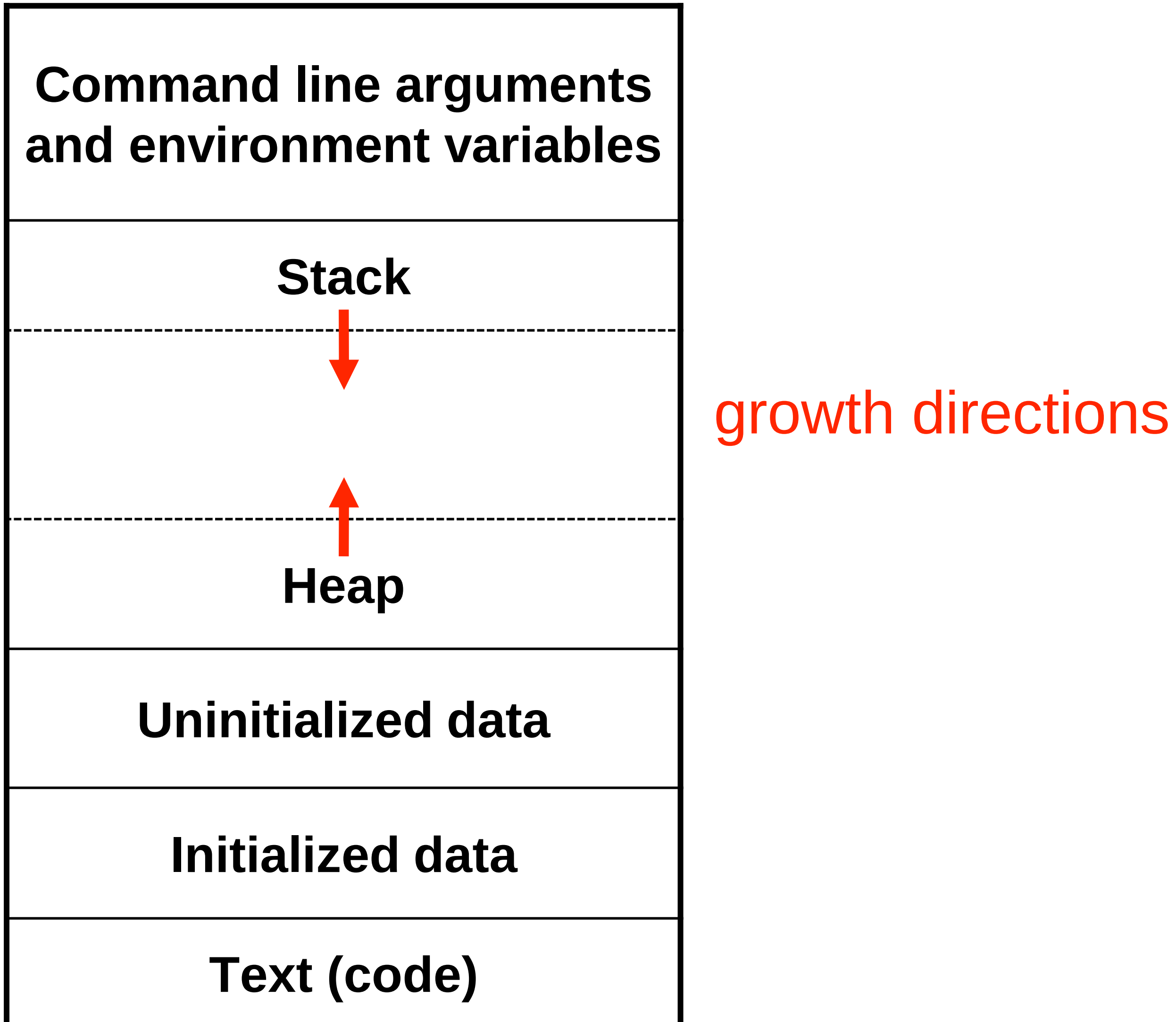
Process Switching

- Before execution, the processor state of a process must be loaded first to the specific processor
- During execution, the processor state of the process changes
- Context switching: a running process stops and another one starts (or resumes)
  - The processor state of the current process is stored
  - The processor state of the next process is loaded

Process Memory

- Each process has its own (private) memory space
  - A process cannot access the memory of another process
  - This provides basic safety in a multi-user environment
- The operating system has full access to the memory of all processes
- Communication between processes is important
  - When they cooperate to solve a single problem
- Operating systems implement several mechanisms for interprocess communication
  - signals, files, pipes, sockets
  - shared memory

Process Memory Layout



Memory Organization (C/C++)

- Text segment
  - Instruction executed by the processor
  - Can be shared between multiple processes
  - Read-only segment
- Initialized data segment
  - Global variables with initial value:

```
    double Pi = 3.1415;
static char message[] = "hello world!";
```

Memory Organization (C/C++)

- Uninitialized data segment
  - Global variables without initial value

```
int result;
double Matrix[512][512];
```

  - The operating system initializes these variables to zero before the execution of the program
- Stack
  - Local variables, function parameters, returned value
- Heap
  - Dynamic memory management (`malloc`, `calloc, new`, …)

Threads

- Thread: an independent stream of instructions that can be scheduled to run as such by the operating system
  - execution sequence within the process
- A process can create multiple threads
  - each thread executes a specific user-defined function
  - main() is the first (primary) thread
- Threads
  - share the memory space of the process they belong to
  - have their own state and some private memory (stack)
  - are cheap to create but difficult to use correctly
  - can run on different processors

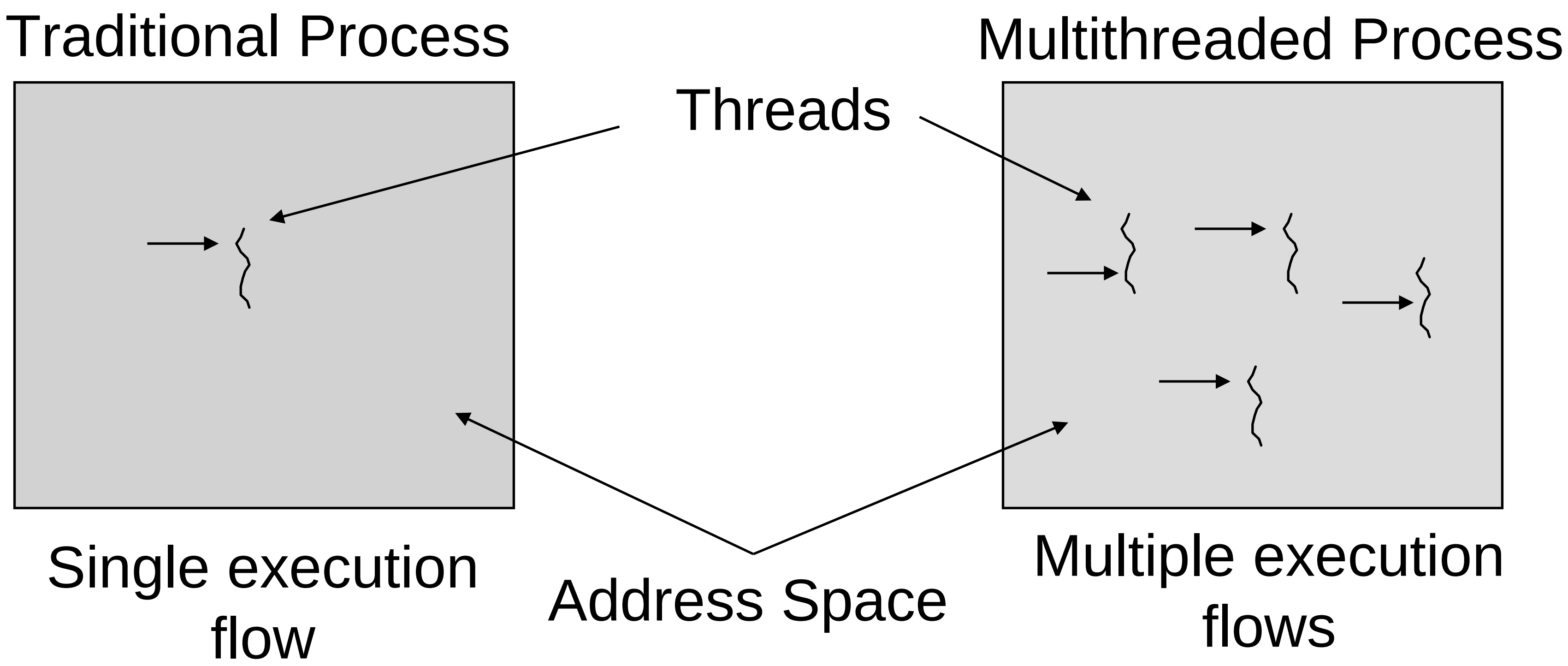
Threads

- Threads: also known as lightweight processes
  - Process: memory, instructions, program counter, stack pointer, registers, file descriptors, …
  - Thread: program counter, stack pointer and stack, registers
- The threads of a process share:
  - Program instructions, most data, open files, signal handlers, current working directory, user and group id
- The threads of a process do not share:
  - Thread id, registers (program counter, stack pointer), stack, errno, signal mask, priority
- POSIX Threads (pthreads): Application Programming Interface (API) defined by the IEEE POSIX.1c standard

Διεργασίες και Νήματα

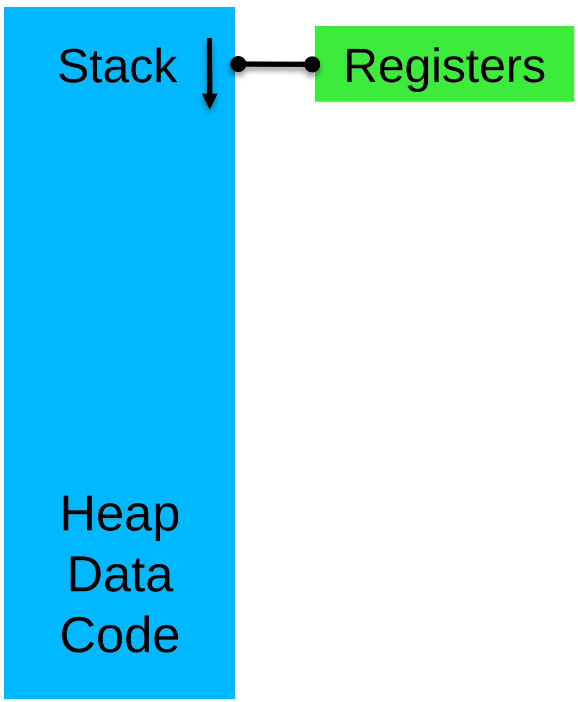
Process ID	***Thread*** **ID**
Program Counter	**Program Counter**
Signal Mask	**Signal Mask**
Registers	**Registers**
Process Priority	***Thread Priority***
Stack Pointer & Stack	**Stack Pointer & Stack**
Heap	***Threads share the memory, heap, signal handlers and file descriptors***
Memory Map	
File Descriptor Table	

Processes and Threads



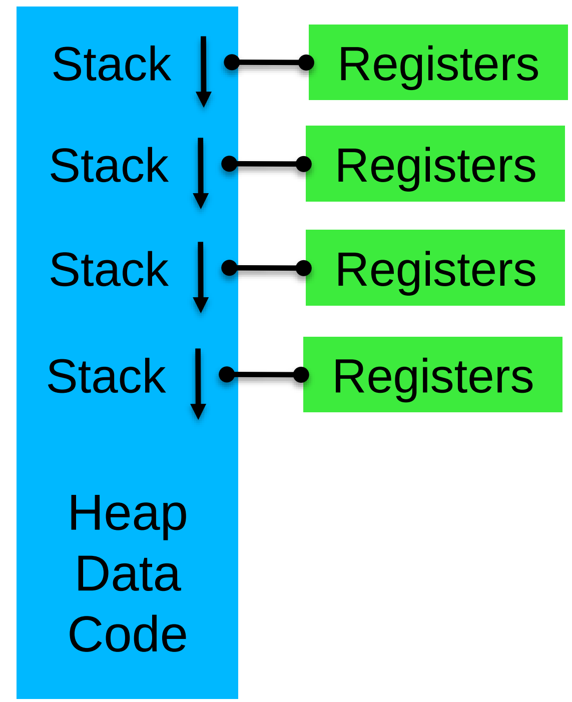
Processes and Threads

Traditional Process



Single instruction execution flow

Multithreaded Process



Multiple instruction execution flows

Processes and Threads

- Advantages/disadvantages of multithreading

  [+] Lower creation and management overhead

  [+] Simpler and cheaper communication between threads than processes

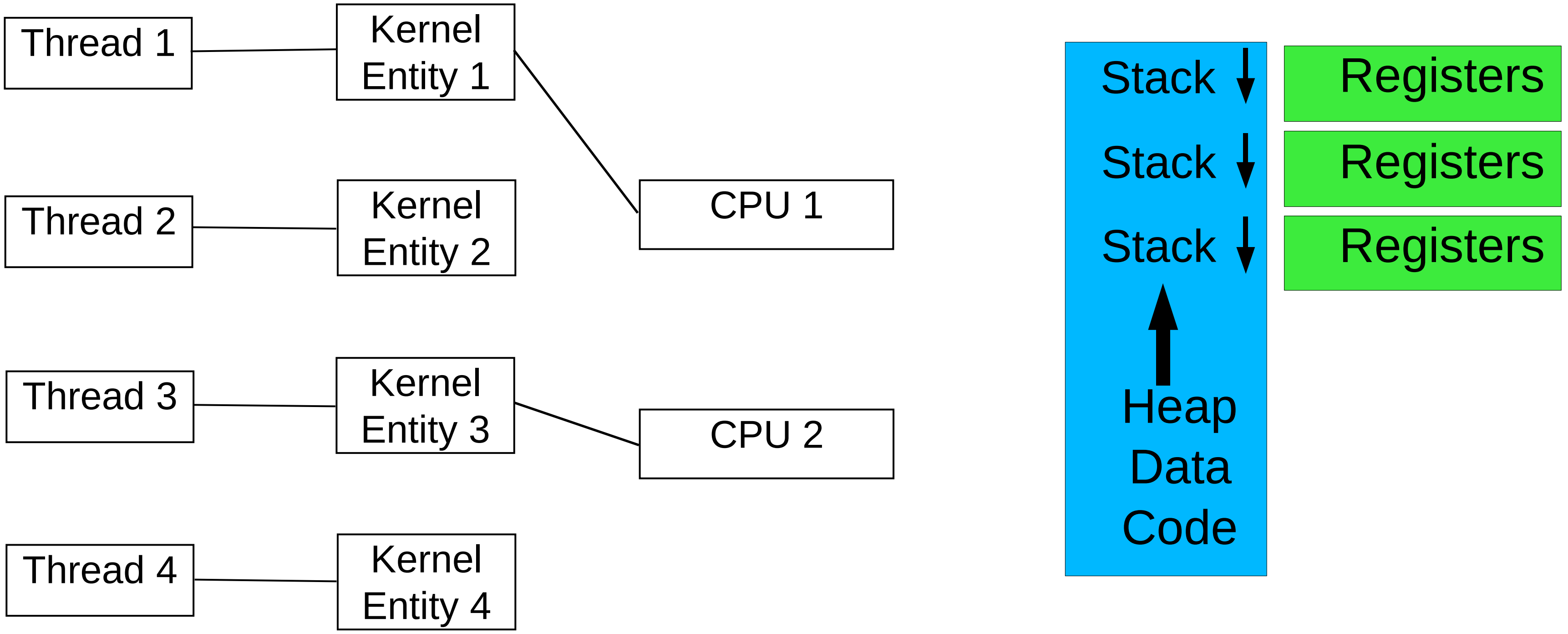
  [-] Error-prone programming
- Thread implementations
  - user level threads
  - system/kernel level threads
- It depends on whether the operating system is aware of the existence of application threads or not

Kernel-level threads

- Implemented as the OS level
  - Each thread is a lightweight process
  - Thread management is based on system calls
- Scheduled by the OS, similarly to processes
  - Straightforward parallel execution of threads on multicore hardware
  - If a threads blocks (at a system call), the rest of the threads (of the same process) continue their execution
- Practically all POSIX Threads implementations follow the specific model

Kernel-level threads

1:1 (one-to-one) model

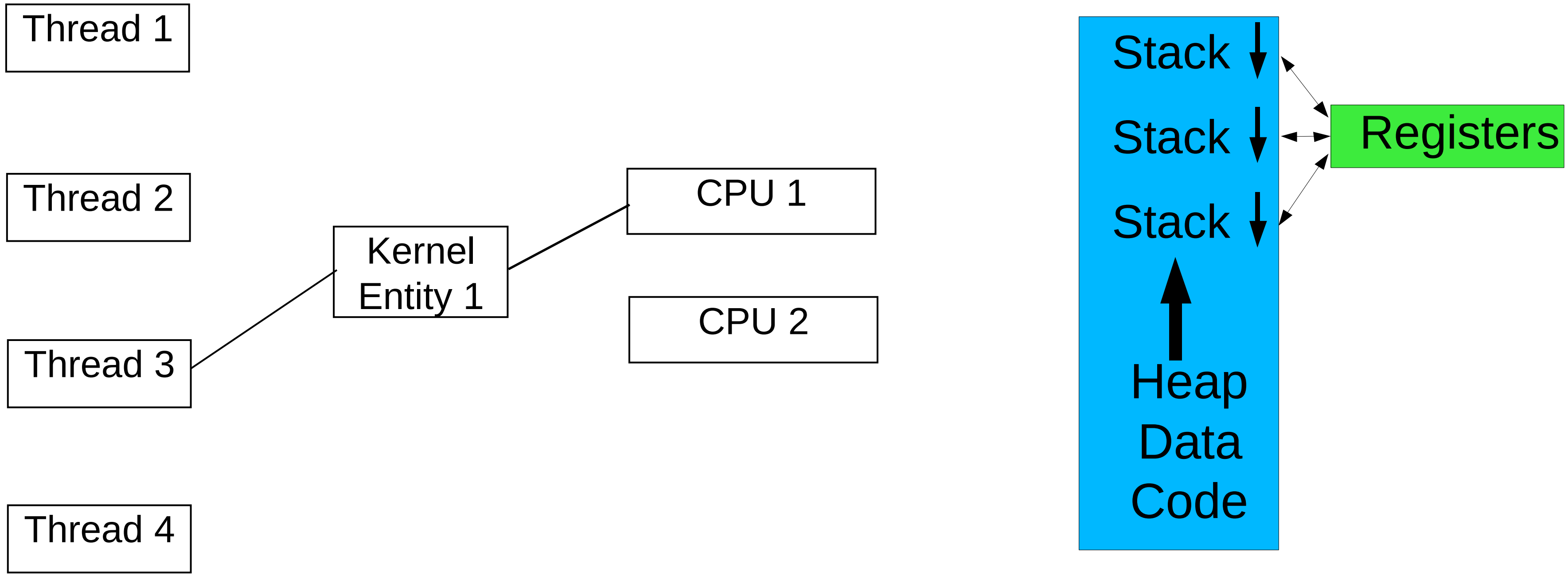


User-level threads

- User-mode implementation
  - Stack and registers
  - Execution management: setjmp, ucontext, assembly, fibers…
- Very fast/lightweight thread management
  - without any system calls
- The threads can be scheduled whenever the owner process is scheduled
  - the runtime system is responsible for their execution.
- User level threads cannot exploit multicore systems because the OS is not aware of them.
- If a thread blocks (e.g., read system call) then the process blocks.

User-level threads

M:1 (many-to-one) model



Context switching

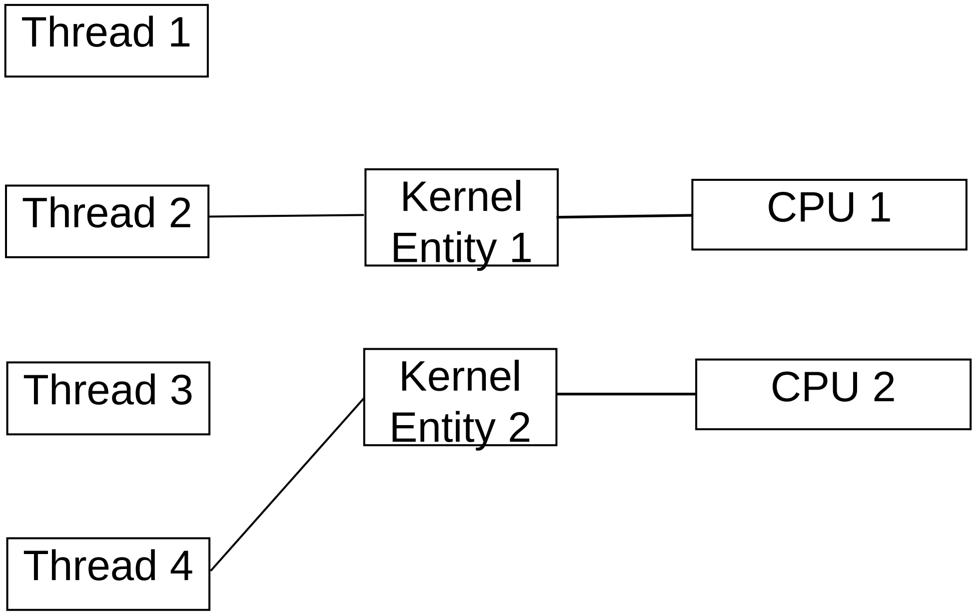
- Context switching can be cooperative (yield-based) or automatic (alarm-based).
- Cooperative or non-preemptive threads
  - When scheduler, it runs to completion without any interruption
  - It can voluntarily release the "processor" (execution flow) and let another thread run
- Preemptive threads
  - The user-level scheduler (see runtime system) can interrupt the execution of a running thread
- Thread state is saved and can be restored later

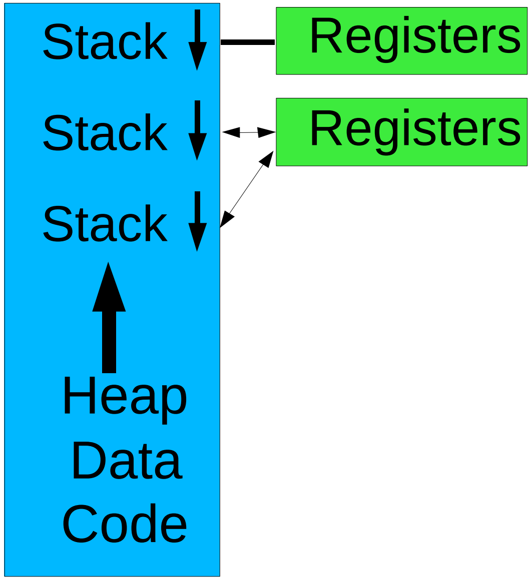
Two-level threads

- Combination of the two previous models
- A single process can create multiple kernel threads and map one or more user level threads to them
- The OS schedules the kernel level threads, while the process (runtime system) schedules the user level threads
- Advanced communication between the application and the runtime system can lead to advanced thread management, e.g., creation of additional threads if there are available computational resources
- Extension of the scheduling algorithm used by the operating system.

Two-level threads

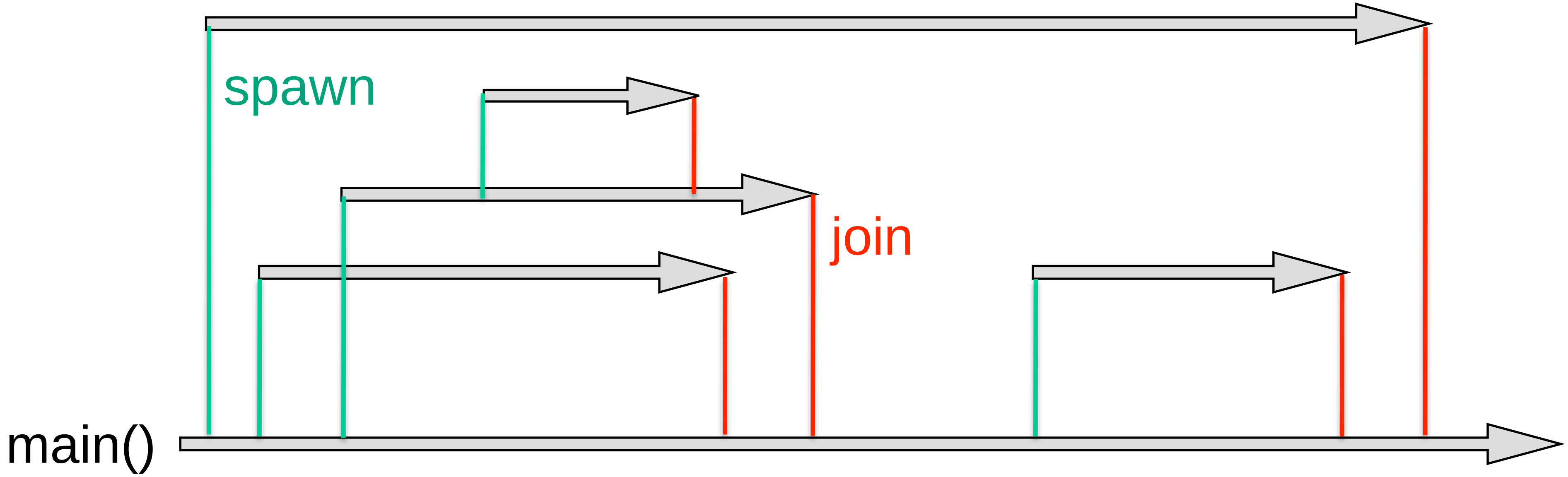
M:N (many-to-many) model



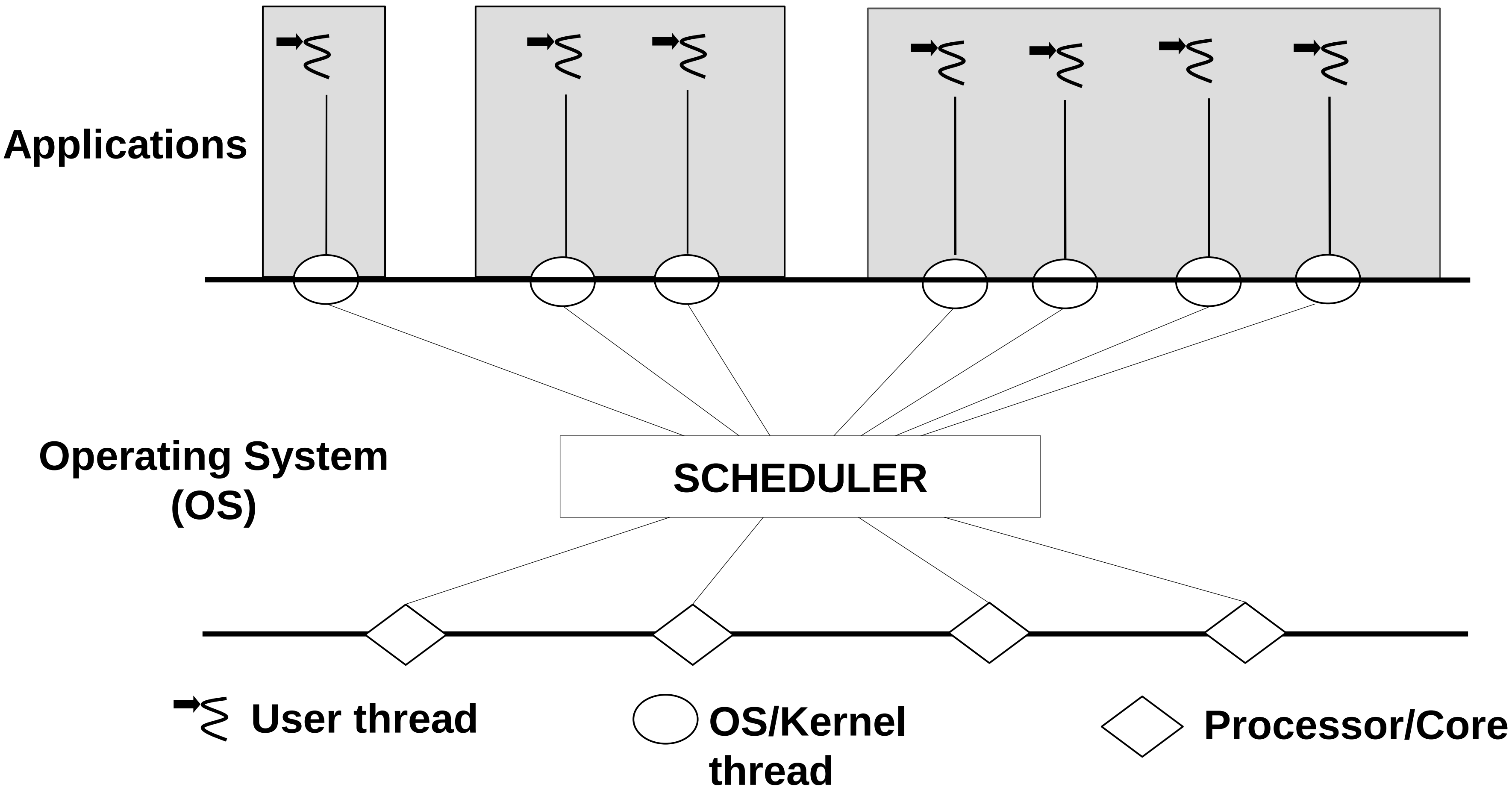


Spawning and Joining Threads

- During the execution of a multithreaded program threads get spawned and joined dynamically



General View



POSIX Threads (Pthreads)

- Standardized C language threads programming interface
  - http://pubs.opengroup.org/onlinepubs/9699919799/

- Header file:

```
#include <pthread.h>
```

- Compilation

```
$ gcc -pthread -o hello hello.c
```

- Execution

```
$ ./hello
```

Skeleton

```
void *func(void *arg)
{
   /* define local data */
   - - - - - - - - - - -
   - - - - - - - - - - -         /* function code */
   - - - - - - - - - - -
   return (void *)&result;
}
```

equivalently:
pthread_exit(&result);

```
main()
{
   pthread_t tid;
   int exit_value;
   - - - - - - - - - - -
   pthread_create (&tid, NULL, func, NULL);
   - - - - - - - - - - -
   pthread_join (tid, &exit_value);
   - - - - - - - - - - -
}
```

Thread Creation

```
int pthread_create (pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*routine)(void *), void *arg);
```

- `thread`: unique identifier for the new thread returned by the subroutine
- `attr`: used to set thread attributes. If NULL, the default values are used.
- `routine`: the C routine that the thread will execute once it is created.
- `arg`: <u>single argument</u> that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.
- if there are no errors, it returns 0

pthread_create

```
#include <pthread.h>

pthread_t tid;
extern void *func(void *arg);
void *arg;

int res = pthread_create(&tid, NULL, func, arg);
```

Passing Multiple Arguments

```
struct data {
  int i;
  float f;
};

void *routine(void *arg) {
  struct data *d = (struct data *) arg;
  int local_i = d->i;
  d->f = 5.0;
  return NULL;
}

int main() {
  pthread_t tid;
  struct data main_data;
  main_data.i = 6;

  pthread_create(&tid, NULL, routine, (void *) &main_data);
  //...
}
```

Thread Joining

```
int pthread_join (pthread_t thread,
                  void **status);
```

- `pthread_join()` blocks the calling thread until the specified thread terminates
- The value returned by the thread function is stored in the memory location specified by `status`
- if there are no errors, it returns 0

pthread_join

```
#include <pthread.h>

pthread_t tid;
int result;


pthread_join(tid, (void *)&result);

pthread_join(tid, NULL);
```

Hello World

ID of calling thread

```
void *work(void *arg)
{
  pthread_t me = pthread_self();
  printf("Hello world from thread %ld!\n", (long)me);
  return NULL;
}

int main(int argc, char **argv)
{
  long i = 1;
  pthread_t thread;

  printf("main thread %ld!\n", (long)pthread_self());

  pthread_create(&thread, NULL, work, (void *)i);
  pthread_join(thread, NULL);

  printf("Child ended, exiting\n");
  return 0;
}
```

Spawning and Joining Threads

```
void *func(void *arg)
{
   sleep(1);
   return NULL;
}

int main(int argc, char * argv[])
{
   pthread_t id[4];

   for (long i = 0; i < 4; i++)    {
       pthread_create(&id[i], NULL, func, NULL);
   }

   for (long i = 0; i < 4; i++)    {
       pthread_join(id[i], NULL);
    }

   return 0;
}
```

Creating and Joining Threads

```
void * func(void * arg)
{
   long sec = (long) arg + 1;
   sleep((long) sec);
   return arg;      /* pthread_exit(arg); */
}

int main(int argc, char * argv[])
{
   pthread_t id[4];
   long result;

   for (long i = 0; i < 4; i++)    {
       pthread_create(&id[i], NULL, func, (void *) i);
   }

   for (long i = 0; i < 4; i++)    {
       pthread_join(id[i], (void *) &result);
       /* result == i */;
   }

   return 0;
}
```

fix: *(long *) arg) +1;

what if we pass &i and also apply the above fix

References

- Advanced Programming in the Unix Environment, W. Richard Stevens
- Programming with POSIX Threads, David R. Butenhof
  - www.openmp.org
- POSIX threads tutorial at LLNL, Blaise Barney
  - https://computing.llnl.gov/tutorials/pthreads/