DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CONCURRENT AND REAL TIME PROGRAMMING

[INQ0091623] AA 2022-23

Lab 5

Parallel Processing

Gabriele Manduchi <gabriele.manduchi@unipd.it>

Andrea Rigoni Garola <andrea.rigonigarola@unipd.it>

Parallel processing

The operating system is able to manage multiple flows of operations at the same time, that is the concurrent execution of multiple programs, even if the underlying computer has a single processor.

The management of multiple tasks on a single processor computer relies on two main facts:

1. **A program does not always require the processor**.

   EXAMPLE: when performing an I/O operation, the processor must await the termination of the data transfer between the device and memory. In the meantime, the operating system can assign the processor to another process.

2. Even in the case where a program is not waiting for I/O, the operating system can decide to reclaim the processor and assign it to another ready program in order to **guarantee the fair execution of all the active processes**.

In this lab we will see two means of implementing parallel processing: **Processes** and **Threads.** Both these entities are handled by the scheduler and, from this point of view, do not differ.

We will see that **the difference between processes and threads lies only in the actions required for the context switch**, which is only a subset of the process-specific information if the processing unit is exchanged among threads of the same process.

The set of active processes (and threads) can be partitioned in two main categories:

• **Ready processes**, that is, processes that could use the processor as soon as it is assigned to them;

• **Waiting processes**, that is, processes that are waiting for the completion of some I/O operation, and that could not make any useful work in the meantime.

Switching among processes

Who is responsible to switch among processes?

The Operative System with the ***Scheduler*** component that supervises the assignment of the processor time to processes.

NOTE:
knowing how the Scheduler component works will be a key factor for the realtime operations in the next lessons.

The transfer of the processor ownership is called ***Context Switch***,

There exists a set of **information that needs to be saved/restored every time the processor is moved from one process to another**.

1. The saved value of the processor registers:
   - the **Program Counter**, that is, the address of the next machine instruction to be executed by the program.
   - the **Stack Pointer**, that is, the address of the stack in memory containing the local program variables and all the arguments of the active procedures of the program at the time the scheduler reclaims the processor.
2. The descriptors of the files and devices currently opened by the program.
3. The virtual memory page tables that are in use for the program.
4. Process-specific data structures maintained by the operating system to manage the process.

Replay

Processes are assigned a priority: higher-priority processes are considered “more important,” and are therefore eligible for the possession of the processor even if other ready processes with lower priority are present. The scheduler organizes ready processes in queues, one for every defined priority, and assigns the processor to a process taken from the non empty queue with the highest priority.

Two main scheduling policies are defined:

1) **First In/First Out (FIFO)**: The ready queue is organized as a FIFO queue, and when a process is selected to run it will execute until it terminates or enters in wait state due to a I/O operation, or a higher priority process becomes ready.

2) **Round Robin (RR)**: The ready queue is still organized as a FIFO queue, but after some amount of time (often called time slice), the running process is preempted by the scheduler even if no I/O operation is performed and no higher priority process is ready, and inserted at the tail of the corresponding queue.

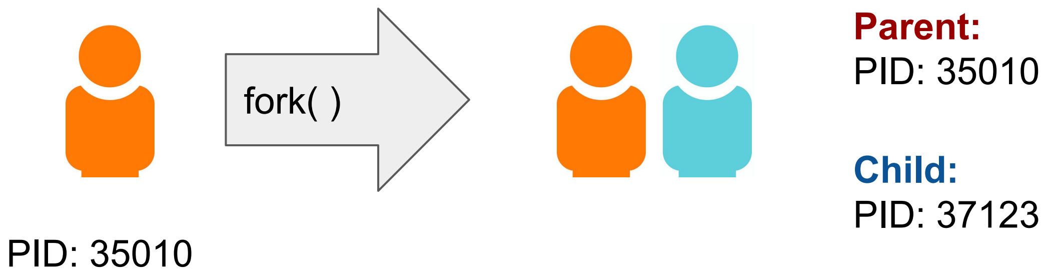
Processes

Process

The API for creating Linux processes is deceptively simple, formed by one system routine with no arguments:

fork()

What fork() actually does is just to create an exact clone of the calling process by replicating the memory content of the process and the associated structures, including the current value of the processor registers.



The child process and the parent process run in separate memory spaces.

Memory Layout of a program

memory assigned to the process is divided into:

- **Stack**, containing the private (sometimes called also automatic) variables and the arguments of the currently active routines. Normally, a processor register is designated to hold the address of the top of the stack;

- **Text**, containing the machine code of the program being executed. This area is normally only read;

- **Data**, containing the data section of the program. Static C variables and variables declared outside the routine body are maintained in the data section;

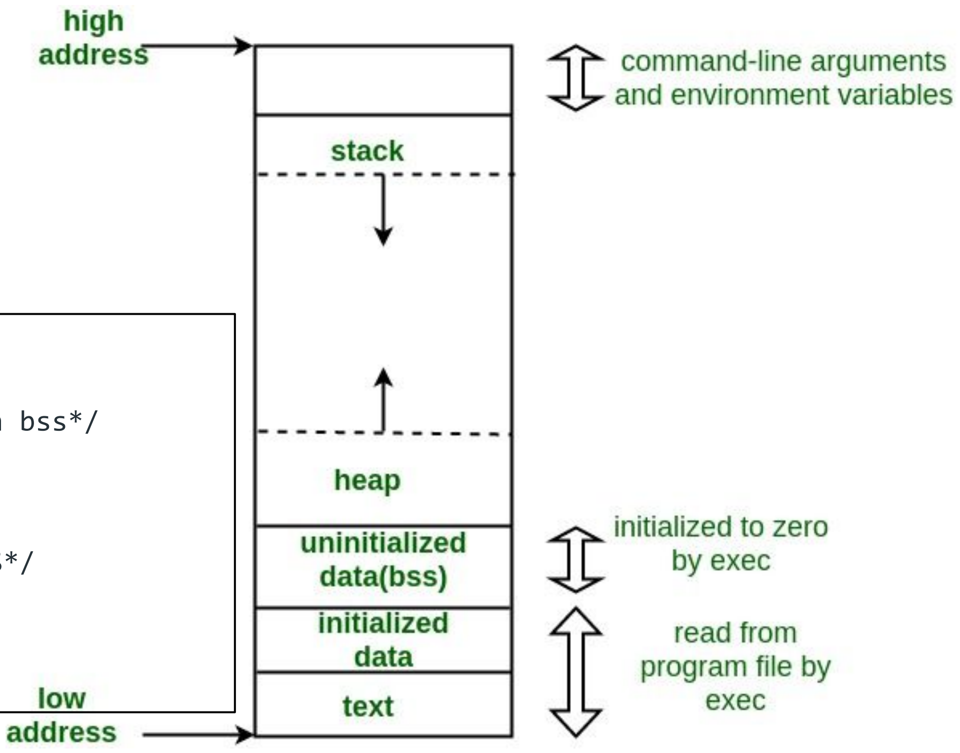
- **Heap**, containing the dynamically allocated data structures. Memory allocated by C malloc() routine or by the new operator in C++ belong to the heap section.

Memory Layout of a program

A typical memory representation of a C program consists of five sections.

1. Text segment (i.e. instructions)
2. Initialized **data** segment
3. Uninitialized data segment (**bss**)
4. Heap
5. Stack

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    /* Initialized static variable stored in DS*/
    static int i = 100;
    return 0;
}
```


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

NOTE:
Linux can give you a hint on the size of a program segments:

```
$ gcc memory-layout.c -o memory-layout
$ size memory-layout

text        data        bss         dec         hex     filename
960          252         12         1224        4c8     memory-layout
```

Simple example to print segments address

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int our_init_data = 30;
int our_noinit_data;

void our_prints(void) {
       int our_local_data = 1;
       printf("\nPid of the process is = %d", getpid());
       printf("\nAddresses which fall into:");
       printf("\n 1) Data  segment = %p", &our_init_data);
       printf("\n 2) BSS   segment = %p", &our_noinit_data);
       printf("\n 3) Code  segment = %p", &our_prints);
       printf("\n 4) Stack segment = %p\n", &our_local_data);
   while(1); // do not stop process
}

int main() {
       our_prints();
       return 0;
}
```

Look for memory segments used by process

CODING EXAMPLE CODING EXAMPLE

```
andrea@HP:~/devel/unipd/crtp/src/lab5$ ./print_segments

Pid of the process is = 1810726
Addresses which fall into:
 1) Data  segment = 0x5653d2903038
 2) BSS   segment = 0x5653d2903040
 3) Code  segment = 0x5653d2900149
 4) Stack segment = 0x7ffdc897e5c4


We can also ask the kernel to print memory segments mapped for this process ...

andrea@HP:/proc/1810726$ cat maps
5653d28ff000-5653d2900000 r--p 00000000 08:11 57545764                   /home/andrea/devel/unipd/crtp/src/lab5/print_segments
5653d2900000-5653d2901000 r-xp 00001000 08:11 57545764                   /home/andrea/devel/unipd/crtp/src/lab5/print_segments
5653d2901000-5653d2902000 r--p 00002000 08:11 57545764                   /home/andrea/devel/unipd/crtp/src/lab5/print_segments
5653d2902000-5653d2903000 r--p 00002000 08:11 57545764                   /home/andrea/devel/unipd/crtp/src/lab5/print_segments
5653d2903000-5653d2904000 rw-p 00003000 08:11 57545764                   /home/andrea/devel/unipd/crtp/src/lab5/print_segments
5653d3dbb000-5653d3ddc000 rw-p 00000000 00:00 0                          [heap]
7f300c95f000-7f300c963000 rw-p 00000000 00:00 0
7f300c963000-7f300c989000 r--p 00000000 08:02 3685186                    /usr/lib/libc-2.33.so
7f300c989000-7f300cad5000 r-xp 00026000 08:02 3685186                    /usr/lib/libc-2.33.so
7f300cad5000-7f300cb21000 r--p 00172000 08:02 3685186                    /usr/lib/libc-2.33.so
7f300cb21000-7f300cb24000 r--p 001bd000 08:02 3685186                    /usr/lib/libc-2.33.so
7f300cb24000-7f300cb27000 rw-p 001c0000 08:02 3685186                    /usr/lib/libc-2.33.so
7f300cb27000-7f300cb30000 rw-p 00000000 00:00 0
 [ … ]
7f300ce68000-7f300ce69000 rw-p 001d8000 08:02 3695107                    /usr/lib/libstdc++.so.6.0.28
7f300ce69000-7f300ce6e000 rw-p 00000000 00:00 0
7f300ce6e000-7f300ce6f000 r--p 00000000 08:02 3685151                    /usr/lib/ld-2.33.so
7f300ce6f000-7f300ce93000 r-xp 00001000 08:02 3685151                    /usr/lib/ld-2.33.so
7f300ce93000-7f300ce9c000 r--p 00025000 08:02 3685151                    /usr/lib/ld-2.33.so
7f300ce9d000-7f300ce9f000 r--p 0002e000 08:02 3685151                    /usr/lib/ld-2.33.so
7f300ce9f000-7f300cea1000 rw-p 00030000 08:02 3685151                    /usr/lib/ld-2.33.so
7ffdc895f000-7ffdc8981000 rw-p 00000000 00:00 0                          [stack]
7ffdc89e3000-7ffdc89e7000 r--p 00000000 00:00 0                          [vvar]
7ffdc89e7000-7ffdc89e9000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

NOTE: You can have also a further information on each segment size in **/proc/PID/smaps** file.

Look for memory segments used by process

Each row in /proc/$PID/maps describes a region of contiguous virtual memory in a process or thread. Each row has the following fields:

```
address            perms  offset    dev    inode  pathname
08048000-08056000  r-xp   00000000  03:0c  64593  /usr/sbin/gpm
```

- **address** - This is the starting and ending address of the region in the process's address space
- **permissions** - This describes how pages in the region can be accessed. There are four different permissions: read, write, execute, and shared. If read/write/execute are disabled, a **-** will appear instead of the r/w/x. If a region is not *shared*, it is *private*, so a p will appear instead of an s. If the process attempts to access memory in a way that is not permitted, a segmentation fault is generated. Permissions can be changed using the mprotect system call.
- **offset** - If the region was mapped from a file (using mmap), this is the offset in the file where the mapping begins. If the memory was not mapped from a file, it's just 0.
- **device** - If the region was mapped from a file, this is the major and minor device number (in hex) where the file lives.
- **inode** - If the region was mapped from a file, this is the file number.
- **pathname** - If the region was mapped from a file, this is the name of the file. This field is blank for anonymous mapped regions. There are also special regions with names like [heap], [stack], or [vdso]. [vdso] stands for virtual dynamic shared object. It's used by system calls to switch to kernel mode.

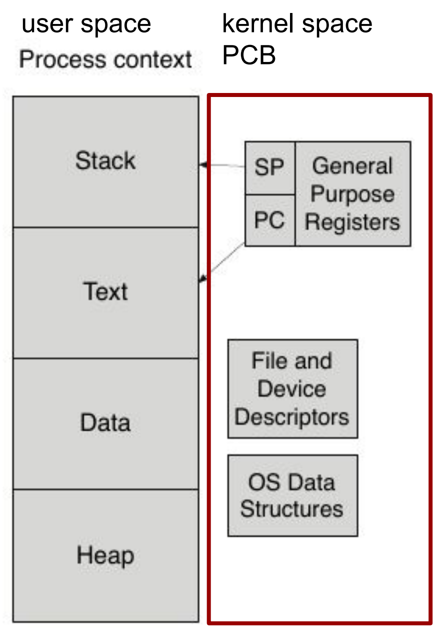
Context switch

Each time a process is removed from access to the processor, sufficient information on its current operating state must be stored such that when it is again scheduled to run on the processor it can resume its operation from an identical position.

pidstat -wt # shows the number of cswch per second

The process state includes all the registers that the process may be using, especially the **program counter**, plus any other operating system specific data that may be necessary. This is usually stored in a data structure called a **process control block** (PCB) or *switchframe*.

The PCB is stored on a per-process stack in **kernel memory** (as opposed to the user-mode call stack)



PCB in LINUX: PROCESS TABLE

In the Linux kernel, each process is represented by a **task_struct** in a doubly-linked list, the head of which is init_task (pid 0, not pid 1). This is commonly known as the process table.

In user mode, the process table is visible to normal users under /proc.

```
/proc/<process-id>/...
```

cmdline:	the command line
sched, stat and **schedstat:**	what the process is currently waiting
environ:	the process environment
uid_map, limits:	process control data for security
maps, smaps:	the memory segments mapped

etc ...

When forks() returns, **two identical processes** at the same point of execution are present.

THE ONLY WAY TO DISTINGUISH THEM IS THE RETURN VALUE !

```
pid = fork();
if(pid == 0)
{
   // CHILD ONLY CODE
   // Actions for the created process
}
else
{
   // PARENT ONLY CODE
   // Actions for the calling process
}
```

When forks() returns **the parent knows that a new clone of himself exists** by the Process ID returned by function… **The child process is unaware** that it comes from a parent process.

NOTE: There is actually a chance to get the parent pid in any case..

```
#include <unistd.h>

pid_t getppid(void);
```

Wait for child termination

The following system routine will suspend the execution of the process until the child process, identified by the process identifier returned by fork(), has terminated.

pid_t wait(pid_t pid, int *status, int options)

What is actually NOT cloned:

At the time of fork() the process behaviour is identical because **their memory is cloned**! So memory **writes**, and **file mappings** (mmap(2)) performed by one of the processes do not affect the other.

The child process is an exact duplicate of the parent process except for the following points:

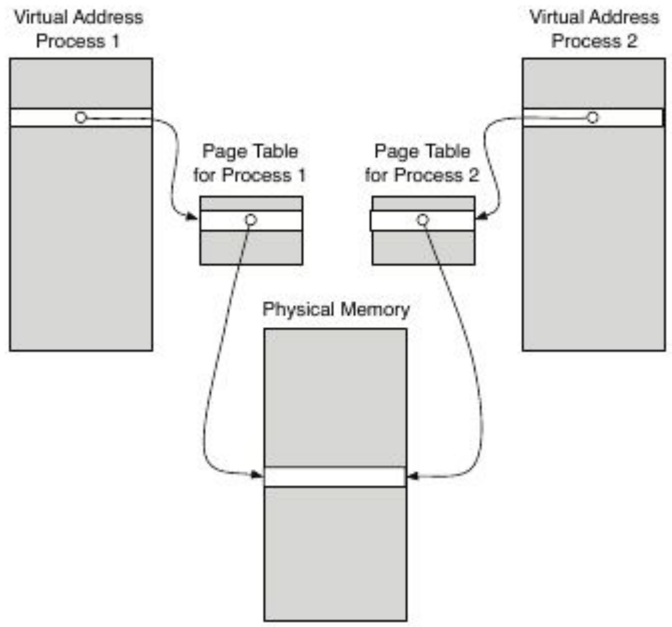
- The child's parent process ID is the same as the parent's process ID.
- Process resource utilizations (getrusage(2)) and CPU time counters (times(2)) are reset to zero in the child.
- The child's set of pending signals is initially empty (sigpending(2)).
- The child does not inherit semaphore adjustments from its parent (semop(2)).
- The child does not inherit timers from its parent (setitimer(2), alarm(2), timer_create(2)).
- The child does not inherit outstanding asynchronous I/O operations from its parent (aio_read(3), aio_write(3)).
- The child does not inherit its parent's memory locks (mlock(2), mlockall(2)).

Share some information among forks

If processes are created to carry out collaborative work, it is necessary that they share memory segments in order to exchange information.

BUT **!**
We know that operating systems supporting **virtual memory** different processes can access to segments of shared memory by setting appropriate values in the page table entries corresponding to the shared memory pages.

We used that for sharing the WebCam frame buffers memory from Kernel space to user space



Shared Memory among forks

The definition of a segment of shared memory is done in Linux in two steps:

1. A segment of shared memory of a given size is created via system routine **shmget()**;

   int shmget(key_t key, size_t size, int shmflg)

   ```
   memId = shmget(key, size, IPC_CREAT | IPC_EXCL);
   ```

   memId: When creating a shared memory segment, it is necessary to provide an unique identifier to it so that the same segment can be referenced by different processes.

2. A region of the virtual address space of the process is "attached" to the shared memory segment via system routine **shmat()**.

   ```
   startAddr = (char *)shmat(memId, NULL, 0666);
   ```

key

Unix requires a key of type **key_t** defined in file **sys/types.h** for requesting resources such as shared memory segments, message queues and semaphores.

There are three different ways of using keys:

1. a specific **integer value** (e.g., 123456 **RISKY !**)
2. a key generated with function **ftok()**
3. a uniquely generated key using **IPC_PRIVATE** (i.e., a private key).

Function ftok() takes a character string that identifies a path and an integer (usually a character) value, and generates an integer of type key_t based on the first argument with the value of id in the most significant position.

Thus, as long as processes use the same arguments to call ftok(), the returned key value will always be the same.

If **IPC_PRIVATE** is used, the system will generate a **unique key** and **guarantee that no other process will have the same key**.

shmget

The system call that requests a shared memory segment is shmget():

```
shm_id = shmget(
            key_t    k,      /* the key for the segment       */
            int      size,   /* the size of the segment       */
            int      flag);  /* create/use flag               */
```

size is the size of the requested shared memory.
flag specifies the way that the shared memory will be used.

For our purpose, only the following two **flag** values are important:

- **IPC_CREAT | 0666** for a server
  (creating and read and write access to the server)
- **0666** for any client
  (granting read and write access to all the clients)

NOTE:

don't change **0666** to 666. The leading 0 of an integer indicates that the integer is an octal number. Thus, 0666 is 110110110 in binary (wr-wr-wr-)

CODING EXAMPLE   CODING EXAMPLE

Shared memory example:

```
#include <sys/ipc.h>
#include <sys/shm.h>
# include <sys/types.h>

/* The numeric identifier of the shared memory segment
  the same value must be used by all processes sharing the segment */
#define MY_SHARED_ID 1

key_t key;        //Identifier to be passed to shmget()
int memId;        //The id returned by shmget() to be passed to shmat()
void *startAddr;  //The start address of the shared memory segment

/* Creation of the key. Routine ftok() function uses the identity
  of the file path passed as first argument (here /tmp is used, but it
  may refer to any existing file in the system) and the least
  significant 8 bits of the second argument */
key_t key = ftok("/tmp", MY_SHARED_ID);

/* First try to create a new memory segment. Flags define exclusive
  creation, i.e. if the shared memory segment already exists, shmget()
  returns with an error */
 memId = shmget(key, size, IPC_CREAT | IPC_EXCL);
 if(memId == -1)  {    /* MEMORY SEGMENT ALREADY CREATED FROM ANOTHER PROCESS */
   /* shmget() is called again without the CREATE option */
   memId = shmget(key, size, 0);
 }
 /* If AGAIN memId == -1 here, an error occurred in the creation of the shared memory segment */
 if(memId != -1) {
 /* Routine shmat() maps the shared memory segment to a range of virtual addresses */
   startAddr = (char *)shmat(memId, NULL, 0666);
 /* From now, memory region pointed by startAddr
    is the shared segment */
// ...
```

process_example

We list here the complete code to improve the sum computation of all numbers in a large matrix of 10e3x10e3 values.

NOTE ON MEMORY ALLOCATION:

Since fork() creates a clone of the calling process including the associated memory, the amount of processing at every child process creation could be very high due to the fact that the main process has allocated in memory the “very large matrix”.

Fortunately this is not the case because the **memory pages in the child process are not actually physically duplicated**. Rather, the corresponding page table entries in the child process refer to the same physical pages of the parent process and are marked as ***Copy On Write***.

process_example

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>

#include <unistd.h> // fork()

#define MAX_PROCESSES 256
#define ROWS 10000L
#define COLS 10000L

/* Arguments exchanged with child processes */
struct argument{
 int startRow;
 int nRows;
 long partialSum;
};
/* The shared memory contains the arguments exchanged between parent
  and child processes and is pointer by processArgs */
struct argument *processArgs;

/* Matrix pointer: it will be dynamically allocated */
long *bigMatrix;

/* The current process index, incremented by the parent process before
  every fork() call. */
int currProcessIdx;
```

process_example

```c
/* Child process routine: make the summation of all the elements of the
  assigned matrix rows. */
static void processRoutine()
{
 int i, j;
 long sum = 0;
 printf("I'm working on process %d\n", currProcessIdx);
/* processArgs is the pointer to the shared memory inherited by the
  parent process. processArg[currProcessIdx] is the argument
  structure specific to the child process */
 for(i = 0; i < processArgs[currProcessIdx].nRows; i++)
   for(j = 0; j < COLS; j++)
     sum += bigMatrix[(processArgs[currProcessIdx].startRow + i) * COLS
                      + j];
/* Report the computed sum into the argument structure */
 processArgs[currProcessIdx].partialSum = sum;
}
```

process_example

```
int main(int argc, char *args[]) {
 int memId;
 long totalSum;
 int i, j, nProcesses, rowsPerProcess, lastProcessRows;
/* Array of process identifiers used by parent process in the wait cycle */
 pid_t pids[MAX_PROCESSES];
/* Get the number of processes from command parameter */
 if(argc != 2) {
   printf("Usage: processs <numProcesses>\n");
   exit(0);
 }
 sscanf(args[1], "%d", &nProcesses);
/* Create a shared memory segment to contain the argument structures
   for all child processes. Set Read/Write permission in flags argument. */
 memId = shmget(IPC_PRIVATE, nProcesses * sizeof(struct argument), 0666);
 if(memId == -1) {
   perror("Error in shmget");
   exit(0);
 }
/* Attach the shared memory segment. Child processes will inherit the shared segment attached */
 processArgs = shmat(memId, NULL, 0);
 if(processArgs == (void *)-1)
 {
   perror("Error in shmat");
   exit(0);
 }
```

process_example

```
/* Allocate  the matrix M */
 bigMatrix = malloc(ROWS*COLS*sizeof(long));
/* Fill the matrix with some values */
 // …
/* If the number of rows cannot be divided exactly by the number of
  processs, let the last thread handle also the remaining rows  */
 rowsPerProcess = ROWS / nProcesses;
 if(ROWS % nProcesses == 0)  lastProcessRows = rowsPerProcess;
 else lastProcessRows = rowsPerProcess + ROWS % nProcesses;
/* Prepare arguments for processes */
 for(i = 0; i < nProcesses; i++) {
   processArgs[i].startRow = i*rowsPerProcess;
   if(i == nProcesses - 1) processArgs[i].nRows = lastProcessRows;
   else processArgs[i].nRows = rowsPerProcess;
 }
/* Spawn child processes */
 for(currProcessIdx = 0; currProcessIdx < nProcesses; currProcessIdx++) {
   pids[currProcessIdx] = fork();
   if(pids[currProcessIdx] == 0)  {
/* This is the child process */
     processRoutine();
     exit(0);
   }
 }
```

process_example

```
/* Wait termination of child processes and perform final summation */
 totalSum = 0;
 for(currProcessIdx = 0; currProcessIdx < nProcesses; currProcessIdx++)  {
/* Wait child process termination */
   waitpid(pids[currProcessIdx], NULL, 0);
   totalSum += processArgs[currProcessIdx].partialSum;
 }


}
```

Threads

Process and threads

Very often, most of the time spent at the context switch is due to saving and restoring the page table since the page table entries describe the possibly large number of memory pages used by the process. Moreover for this reason, creating new processes involves the creation of a large set of data structures.

A new model of computation could ease this issue: **the threads**.

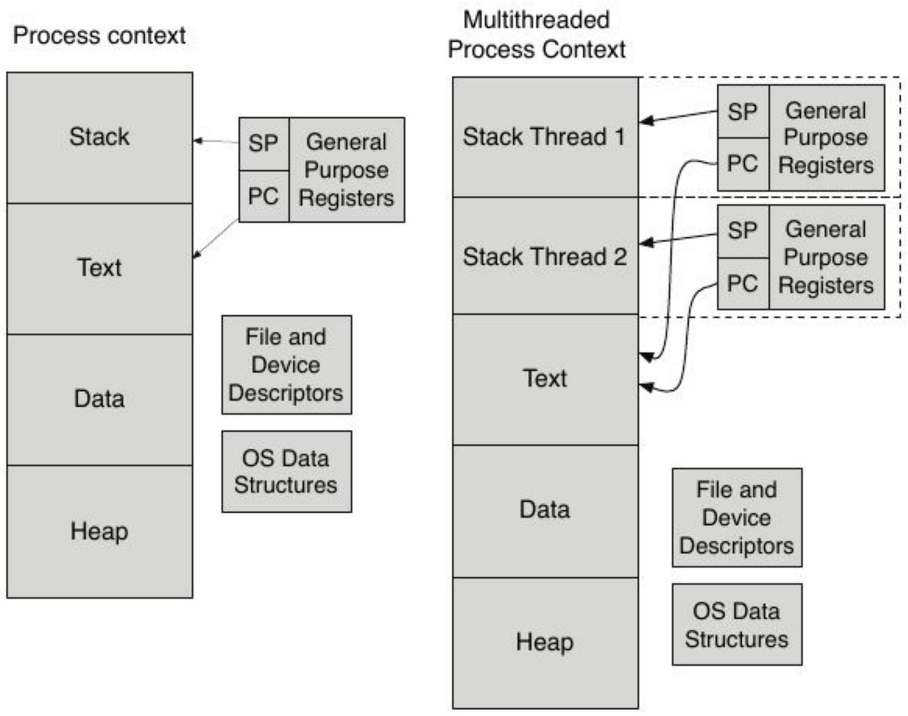
Conceptually, threads are not different from processes because both entities provide an independent flow of execution for programs.

This means that all the problems, strategies, and solutions for managing concurrent programming apply to processes as well as to threads.

There are, however, several important differences due to the amount of information that is saved by the operating system in context switches. Threads, in fact, live in the context of a process and share most process-specific information, in particular memory mapping. This means that the threads that are activated within a given process share the same memory space and the same files and devices. For this reason, **threads are sometimes called “*lightweight processes*”**.

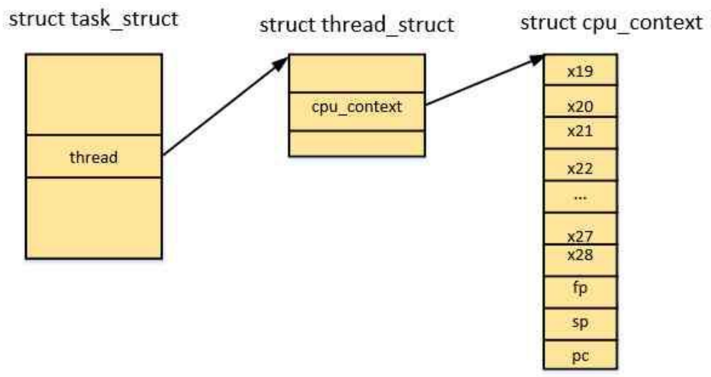
Threads context switch

The context switch for threads in linux is handled by a specific attribute of the PCB (i.e. the task_struct we saw).



Process context
Stack
Text
Data
Heap
SP
PC
General Purpose Registers
File and Device Descriptors
OS Data Structures
Multithreaded Process Context
Stack Thread 1
Stack Thread 2
Text
Data
Heap
SP
PC
General Purpose Registers
SP
PC
General Purpose Registers
File and Device Descriptors
OS Data Structures

GNU-Linux kernel structures



```
#include <sched.h>
```

https://docs.huihoo.com/doxygen/linux/kernel/3.7/structtask__struct.html

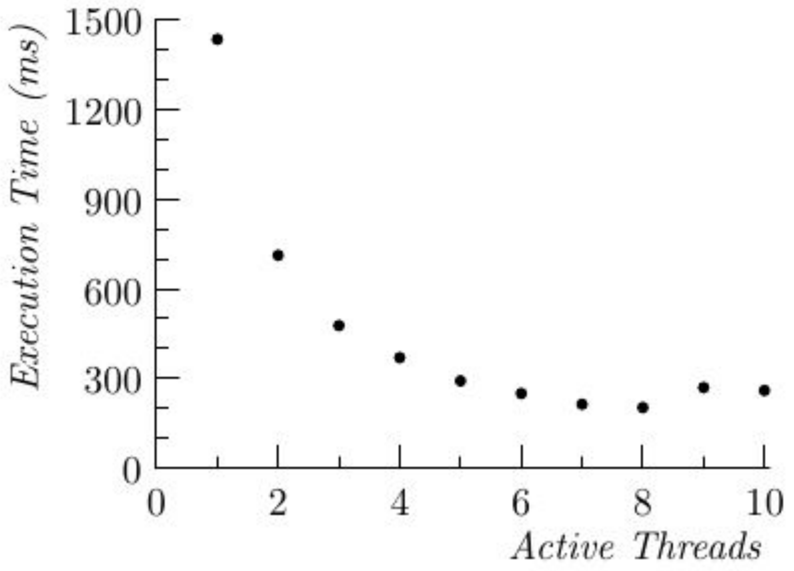
Example of bigMatrix sum

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define MAX_THREADS 256
#define ROWS 10000
#define COLS 10000

/* Arguments exchanged with threads */
struct argument{
 int startRow;
 int nRows;
 long partialSum;
} threadArgs[MAX_THREADS];

/* Matrix pointer: it will be dynamically allocated */
long *bigMatrix;

/* Thread routine: make the summation of all the elements of the assigned matrix rows */
static void *threadRoutine(void *arg)
{
 int i, j;
/* Type-cast passed pointer to expected structure
   containing the start row, the number of rows to be summed
   and the return sum argument */
 struct argument *currArg = (struct argument *)arg;
 long sum = 0;
 for(i = 0; i < currArg->nRows; i++)
   for(j = 0; j < COLS; j++)
     sum += bigMatrix[(currArg->startRow + i) * COLS + j];
 currArg->partialSum = sum;
 return NULL;
}
```

Example of bigMatrix sum

CODING EXAMPLE CODING EXAMPLE

```
int main(int argc, char *args[]) {

 pthread_t threads[MAX_THREADS];
 long totalSum;
 int i, j, nThreads, rowsPerThread, lastThreadRows;

 /* Allocate  the matrix M */
 bigMatrix = malloc(ROWS*COLS*sizeof(long));

 /* If the number of rows cannot be divided exactly by the number of threads, let the last thread handle also the remaining rows */
 rowsPerThread = ROWS / nThreads;
 if(ROWS % nThreads == 0)
   lastThreadRows = rowsPerThread;
 else
   lastThreadRows = rowsPerThread + ROWS % nThreads;

/* Prepare arguments for threads */
 for(i = 0; i < nThreads; i++) {
   /* Prepare Thread arguments */
   threadArgs[i].startRow = i*rowsPerThread;
   if(i == nThreads - 1)
     threadArgs[i].nRows = lastThreadRows;
   else
     threadArgs[i].nRows = rowsPerThread;
 }
 /* Start the threads using default thread attributes */
 for(i = 0; i < nThreads; i++)
   pthread_create(&threads[i], NULL, threadRoutine, &threadArgs[i]);

 /* Wait thread termination and use the corresponding sum value for the final summation */
 totalSum = 0;
 for(i = 0; i < nThreads; i++) {
   pthread_join(threads[i], NULL);
   totalSum += threadArgs[i].partialSum;
 }
}
```



Execution Time (ms)
1500
1200
900
600
300
0
0
2
4
6
8
10
Active Threads

pthreads

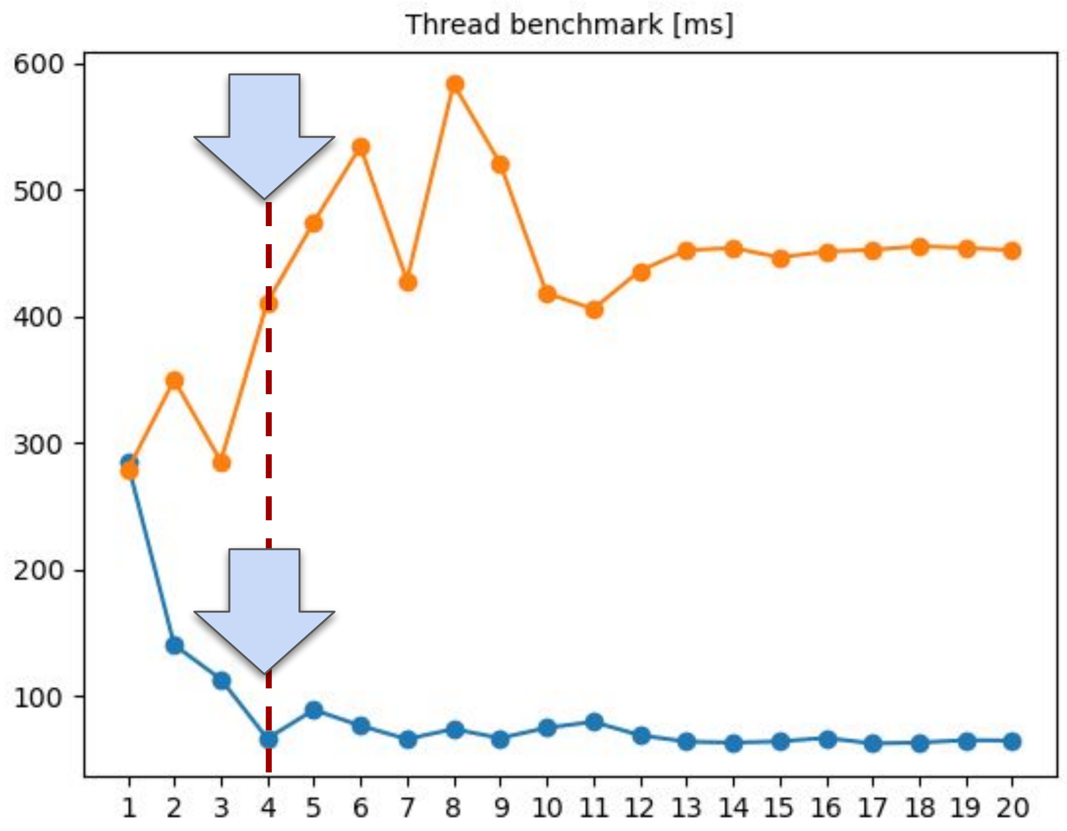
```
pthread_t threads[N];
```

```
pthread_create(&threads[i], NULL, start_routine, &args);
```

```
#include <pthread.h>

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

```
pthread_join(threads[i], NULL);
```

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



●—●

Total CPU time consumed
On all threads

`CLOCK_PROCESS_CPUTIME_ID`

●—●

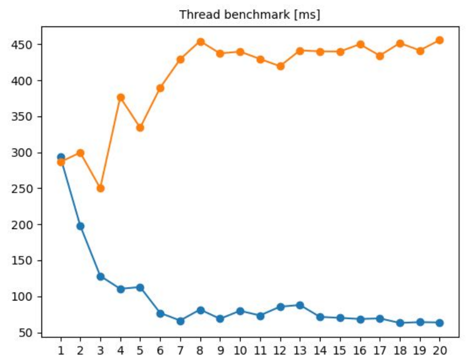
Wall time clock elapsed

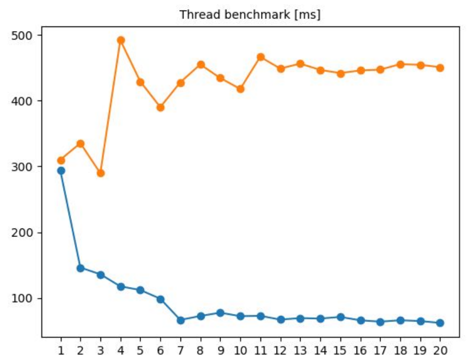
`CLOCK_REALTIME`

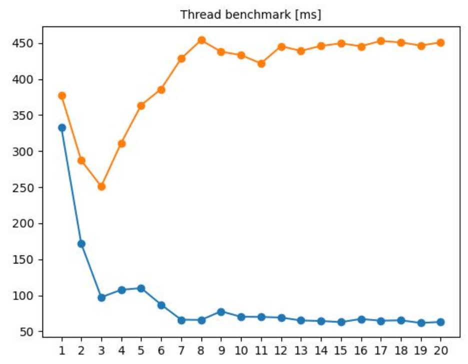
Non deterministic timing of thread execution !

SMP systems may have processors with different speeds. The cpu timer (TSC or ITC) may suddenly encounter a cpu timer running at different speeds if a process is moved to a different cpu. Glibc cannot handle that situation.

Systems may reduce the clock speed and therefore vary the speed of the CPU counter. Glibc has no awareness of this.







1. Initial conditions matter
• measured time may depend on state of machine when timing starts

2. Resolution and accuracy of timer
• granularity of your measuring device
• spread in measurements

3. “Heisenberg effect”
• measurement may change quantity you are measuring

4. Compiler optimizations
• may need to look at actual assembly code to make sure compiler has not modified your code in unexpected ways

5. Context-switching by O/S and hardware interrupts
• you may end up measuring stuff outside your code

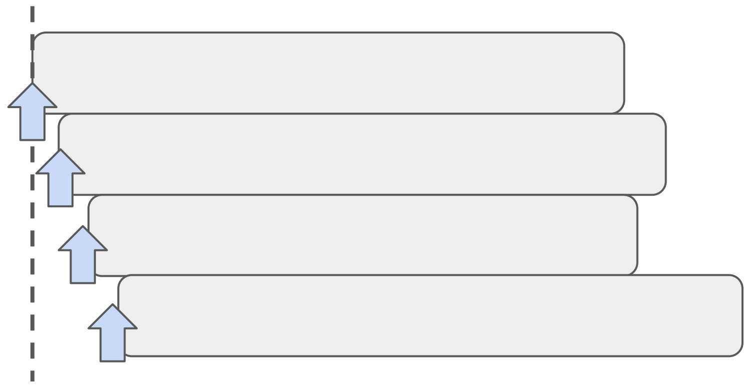
6. Out-of-order execution of instructions
• what you measure may not be what you think you are measuring

Thread concurrency - race hazard

As well as the timing for the overall threads operation, also the single thread suffer the non-deterministic time behavior. And this eventually can become an hazard if threads depends on the others.

VERY SIMPLE EXAMPLE:

```
/* Thread routine: make the summation of all the elements of the  assigned matrix rows */
static void *threadRoutine(void *arg) {
 int i, j;
 struct argument *currArg = (struct argument *)arg;
 long sum = 0;
 printf("I'm working on thread %d\n", currArg->threadN);
 for(i = 0; i < currArg->nRows; i++)
   for(j = 0; j < COLS; j++)
     sum += bigMatrix[(currArg->startRow + i) * COLS + j];
 currArg->partialSum = sum;
 return NULL;
}
```



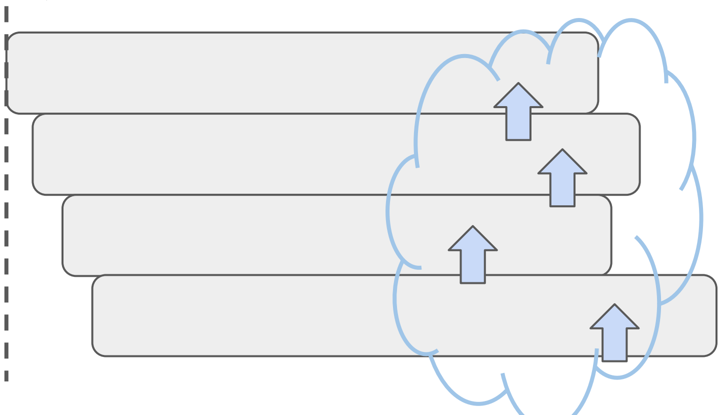
```
$ ./threads_example 10
I'm working on thread 0
I'm working on thread 1
I'm working on thread 2
I'm working on thread 3
I'm working on thread 4
I'm working on thread 5
I'm working on thread 6
I'm working on thread 7
I'm working on thread 8
I'm working on thread 9
```

Thread concurrency - race hazard

As well as the timing for the overall threads operation, also the single thread suffer the non-deterministic time behavior. And this eventually can become an hazard if threads depends on the others.

VERY SIMPLE EXAMPLE:

```
/* Thread routine: make the summation of all the elements of the  assigned matrix rows */
static void *threadRoutine(void *arg) {
 int i, j;
 struct argument *currArg = (struct argument *)arg;
 long sum = 0;
 for(i = 0; i < currArg->nRows; i++)
   for(j = 0; j < COLS; j++)
     sum += bigMatrix[(currArg->startRow + i) * COLS + j];
 printf("I'm working on thread %d\n", currArg->threadN);
 currArg->partialSum = sum;
 return NULL;
}
```



```
$ ./threads_example 10
I'm working on thread 5
I'm working on thread 6
I'm working on thread 4
I'm working on thread 3
I'm working on thread 2
I'm working on thread 0
I'm working on thread 1
I'm working on thread 9
I'm working on thread 7
I'm working on thread 8
```

Looking at threads

If we want to look at threads execution from command line:

EXAMPLE:
(add a pause during execution)

```c
#include <unistd.h> // sleep
#define PAUSE sleep(60);
static void *threadRoutine(void *arg)
{
 int i, j;
 struct argument *currArg = (struct argument *
 long sum = 0;
 printf("I'm working on thread %d\n", currArg-
 for(i = 0; i < currArg->nRows; i++)
   for(j = 0; j < COLS; j++)
     sum += bigMatrix[(currArg->startRow + i)
 currArg->partialSum = sum;
 PAUSE
 return NULL;
}
```

```
$ ./threads_example 10
I'm working on thread 0
I'm working on thread 1
I'm working on thread 2
...
```

```
$ ps -x | grep threads_example
1550977 pts/2    Sl+    0:00 ./threads_example 10
1551010 pts/8    S+     0:00 grep --color=auto threads_exampl
```

```
$ ps -T -p 1550977
    PID    SPID TTY      TIME CMD
1550977 1550977 pts/2    00:00:00 threads_example
1550977 1550978 pts/2    00:00:00 threads_example
1550977 1550979 pts/2    00:00:00 threads_example
1550977 1550980 pts/2    00:00:00 threads_example
1550977 1550981 pts/2    00:00:00 threads_example
1550977 1550982 pts/2    00:00:00 threads_example
1550977 1550983 pts/2    00:00:00 threads_example
1550977 1550984 pts/2    00:00:00 threads_example
1550977 1550985 pts/2    00:00:00 threads_example
1550977 1550986 pts/2    00:00:00 threads_example
1550977 1550987 pts/2    00:00:00 threads_example
```

Threads or Processes - ROUND 1

We have seen that in Linux there are two classes of entities able to execute programs.

Threads have been introduced later as a lightweight version of processes, and are preferable to processes for two main reasons:

1. **Efficiency:** context switch in threads belonging to the same process is fast when compared with context switch between processes. More information, including the page table, has to be saved and restored at every context switch.

2. **Simplicity:** sharing memory among threads is trivial, it suffices to use static variables. We don't need to pass through an explicit query of TLB mmap from kernel.

THREADS WINS THIS ROUND

Seems that threads are the best option so far ! But it's not all sunshine and puppy dogs as we will see...

EXERCISE

update your code with:

```
$ git pull origin master
```

in **src/lab5** there are a set of programs shown during the lab.

Likewise what we did in *threads_benchmark.c* try to build a benchmark of the parallel processing for bigMatrix sum introduced by *process_example.c*