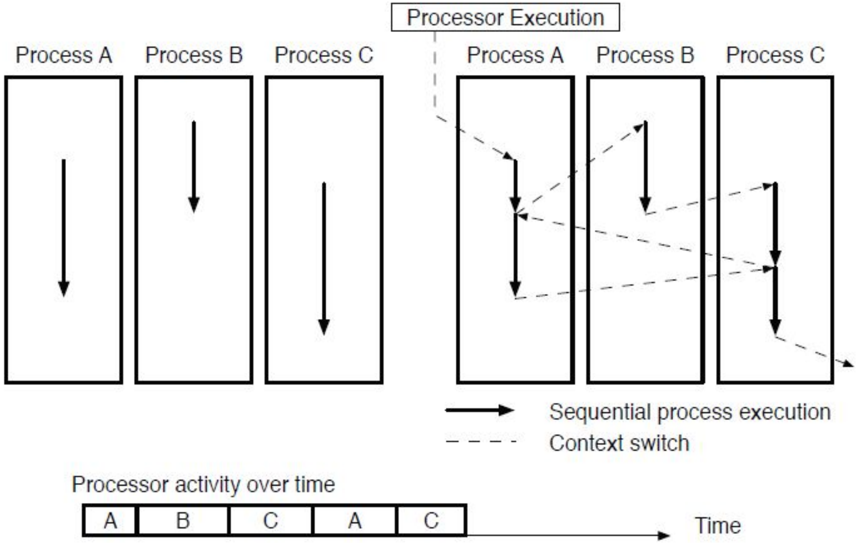
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Processes, Threads and Scheduling

- Process definition
- Process Context
- Process states
- Threads
- Scheduling principles
- Linux scheduler
- Fixed and Dynamic priorities

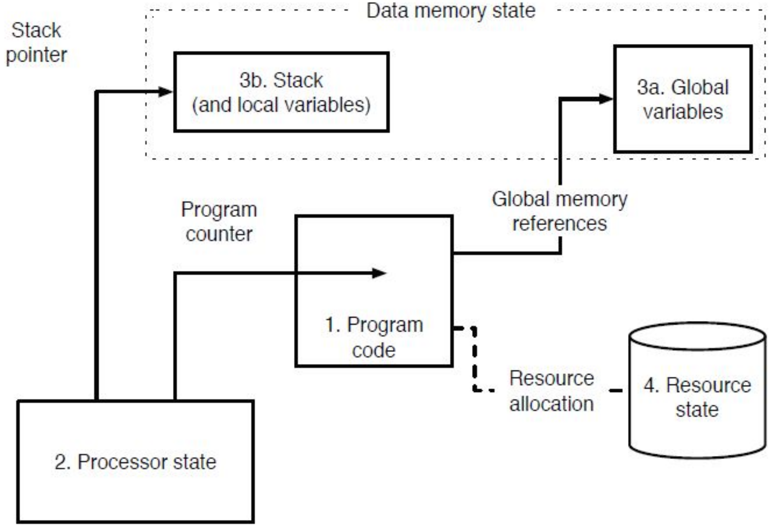
Handling multiple programs via processes

- The process abstraction is the base mechanism to allow concurrent execution of multiple programs on a single processor
- Modern machine host multicore processors and therefore the computer activity is a mix of true parallel and OS managed multiprogramming
- In the following the points below shall be addressed:
    - What actions are required to transfer processor control from one program to another
    - The lifeline of a process
    - How the OS can take control of the processor in order to schedule processes
    - What factors are considered in the choice of the running process



Process Context

- A running program brings a set of information that change over time
- A snapshot of the associated information must be saved by the OS when the program loses processor ownership
- Process context include:
  - The program and the other memory contents managed by the program, in particular the program stack and the global variables
  - The current values of the processor registers, including the PC that holds the address of the next instruction to execute
  - The OS resources currently used by the process (e.g. open files)
- Process context must not be confused with the interrupt context



How to save memory contents

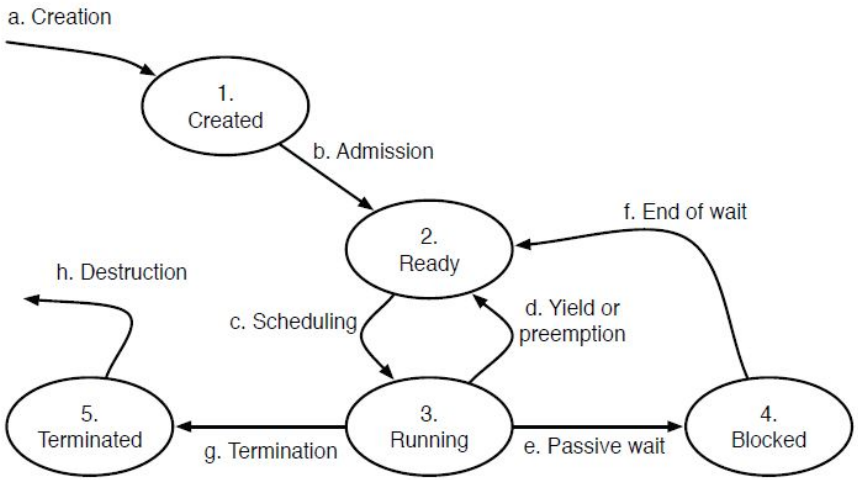
- Registers and OS resources are saved in a data structures owned by the OS called the **Process Control Block (PCB)**
- It would be however not possible to copy the content of the memory used by the process, and memory contents are indeed left where
- It is therefore necessary that physical memory location are independent from memory location as seen by the process (virtual addresses)
- In this case the Page Table entries (possibly referring to swapped pages) used by the process are saved in the PBC
- This holds for
  - The program code
  - The program stack for local variables
  - The static memory content for global and static variables
  - The memory dynamically allocated in the Heap
- The amount of information to be copied in the PCB and be significant is the process is using a large amount of memory and potentially affect the speed of the process context
- TLB must be flushed upon context switch

Memory protection

- The use of virtual memory ensures protection against wrong memory access in process code
- Without virtual memory, user programs may harm other processes- memory or even worse OS data structures
- Using Virtual memory, the process is given a memory area (the pages mapped in the corresponding PTEs) and any access within this area is legal (including program bugs)
- If the process accesses a location whose address is not mapped, an exception (interrupt-like) is generated by the MMU HW, OS regains control and normally the process is aborted
- In any case, OS and other process memory integrity is preserved

The Process states

- A process us ready (computable) when it is able to execute program instructions (i.e. it is not waiting for an I/O operation to conclude)
- Eventually, the scheduler will assign a processor to this task and the its state changes to 'Running'. i.e. the associated program is in execution. **NOTE**: when the program is running the Processor is totally under its control, no OS control at all.
- When performing an I/O operation or, more in general, requesting any OS function, the process may enter wait state because it is waiting the termination of an action (e.g. I/O operation) and could not use the processor meanwhile
- A process may be forced to return in ready state, i.e. lose the processor against its will. In this case the OS takes control of the processor thanks to an Interrupt.



Threads

.

- In many applications, there are several distinct concurrent activities that are nonetheless related to each other, for example, because they have a common goal
- In this case, implementing them as distinct processes may be difficult because the different processes must share resources such as memory structures and open files
- It may therefore be useful to manage all these activities as a group and share system resources, such as files, devices, and network connections
- This can be done conveniently by envisaging multiple flows of control, or **threads**, within a single process.
- As an added bonus, all of them will implicitly refer to the same address space and thus share memory. This is a useful feature because many interprocess communication mechanisms are indeed based on shared variables.

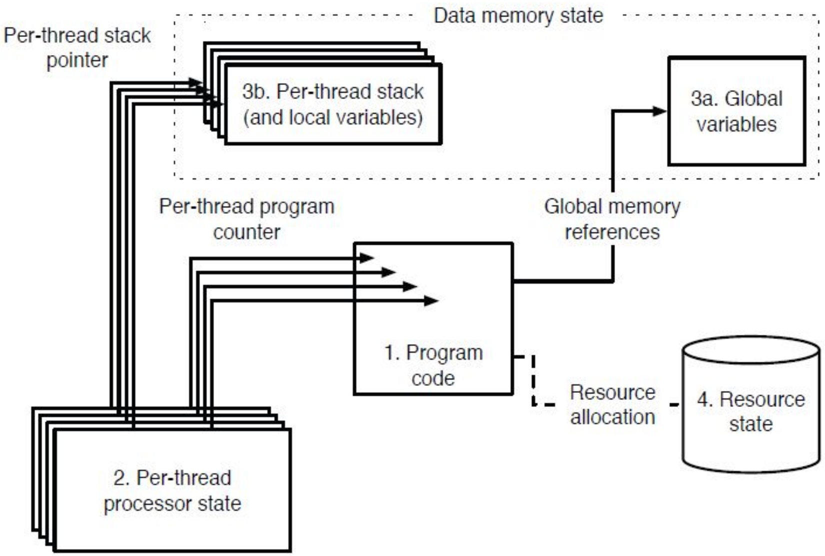
Thread Context

- Threads within the same process share information such as the page table (same virtual address space) and open files

- Other information is thread specific - every thread is running a different program (in practice different routines of the same main program)

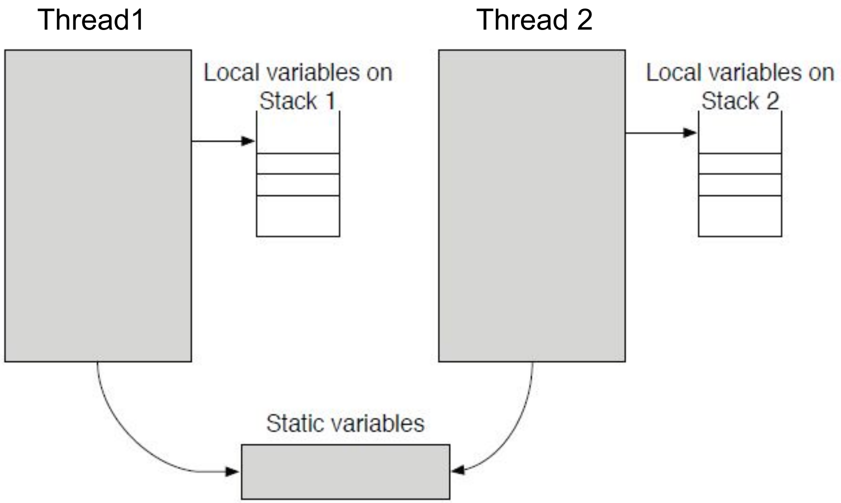
- This information include:
    - Processor registers, including the Program Counter (PC)
    - Local program variables, maintained in a pre-thread stack

- Other memory contents such as Global Variables (static) and the program code itself are shared



The Thread Memory Model

- Variables allocated on the (per thread) stack are those declared as local variable in the C routines
- Different threads can execute the same routine without affecting each other local variables
- Variables declared outside routines and as static in the C code are shared among threads
- This offers an easy way for sharing memory structures among threads in the same process
- HOWEVER, shared memory alone is in general not enough to ensure correct communication and synchronization among threads

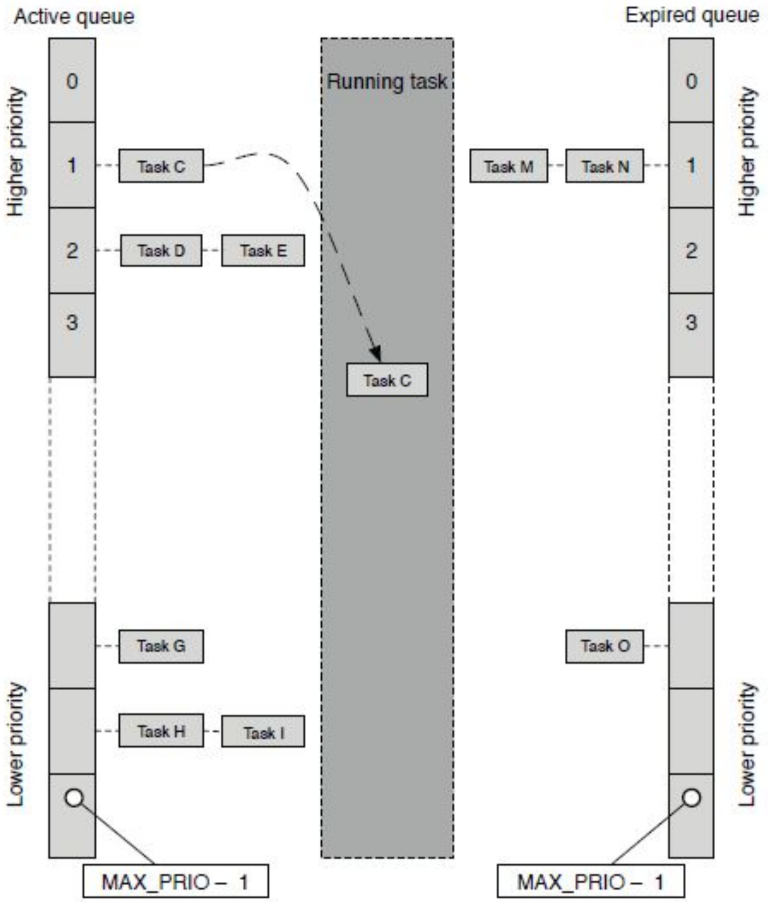


The Scheduler

- The scheduler is the OS component that supervises the selection of the process or thread to become running
- Threads and processes play an equivalent role and in the following they shall be referred as **tasks**
- The selection of the task to become running is based on its **priority**
- The ready task at the highest priority shall be selected
- Scheduler action may be requested when
  - An interrupt occurs, possibly changing the state of a pending I/O operation and making a waiting process ready
  - Any process issues a OS call (e.g. I/O operation) possibly changing the state of the calling process from running to wait.
- The involved OS code shall make a call to **`schedule()`** at the end of the corresponding action
- An important interrupt source is the Timer Interrupt, issued normally at a rate of 60 Hz
  - It shall update dynamic priorities and time slices

Scheduler data structures

- The scheduler organizes ready tasks in queues based on their priority
- Every task that is not declared as FIFO task is assigned a time slice in order to ensure fairness among tasks of the same (highest) priority
- **`scheduler_tick()`** is called at every timer interrupt. It decreases the current time slice of the currently running task
- Whenever time time slice reaches zero, the task is moved to the expired queue and the processor will be assigned next task in the active queue will be assigned (via a call to **`schedule()`**)
- If The active queue is empty for a given priority , it shall be swapped with the corresponding expired queue
- Selection of the highest priority queue is performed in O(1) time using a bitmap



Task Priority

- In Linux 140 priority levels are defined (lower number -> higher priority)
- Priorities 0 to 100 are fixed the others are dynamic
- The remaining priorities can be dynamically changed by the OS based on a given nice value.
- Dynamic priority adjustment aims at providing improved fairness
    - In practice, a more ***fluid*** user interaction
- Priority adjustment is carried out during Timer interrupts, decreasing a counter associated with the currently running task
- When the counter reaches 0 the priority of the task is lowered
- With a similar mechanism the priority of the waiting tasks is increased
- The rationale behind is to let tasks that tend to use less CPU should be given a higher priority.
- In this way the computer shall not be blocked even if the highest priority task is running an infinite loop.

Fixed vs Dynamic task priorities

- Dynamic task priority is important in improving user interaction, avoiding annoying blocks in task execution (because of a CPU consuming higher priority task)

- On the other side, it is not possible to ensure that an important task to which the highest priority has been assigned will retain its priority
    - As a consequence its **latency** may increase

- In the following we shall define latency of a given task as the time between the instant in which an event occurs that makes the task ready and the time the task gains processor ownership

- The event may be:
    - The termination of an I/O operation
    - The availability of a new input
    - The termination of a given interval (cyclic tasks)
    - The occurrence of an interrupt signaling a condition to be served

- Several factors affect task latency, among which:
    - The presence of tasks at higher or equal priority
    - The number of available cores
    - The OS latency