Scuola Superiore Sant'Anna

Real-Time System Laboratory

Operating Systems
Processes and Threads

Giuseppe Lipari

Process

- The fundamental concept in any operating system is the “process”
  - A process is an executing program
  - it has its own set of resources (memory, file descriptors, etc.)
  - An OS can execute many processes at the same time (concurrency)

Process traces

- When a computer executes a sequential process, it goes through states
- At each step, the state of the process is changed
- State:
  - Set of processor registers
  - Set of process variables

Process History

- A program for computing GCD

```
int gcd(int a, int b)
{
   while (a!=b) {
      if (a < b) b = b-a;
      else a = a – b;
   }
   return a;
}
```

Steps	a	b
1	21	15
2	6	15
3	6	9
4	6	3
5	3	3

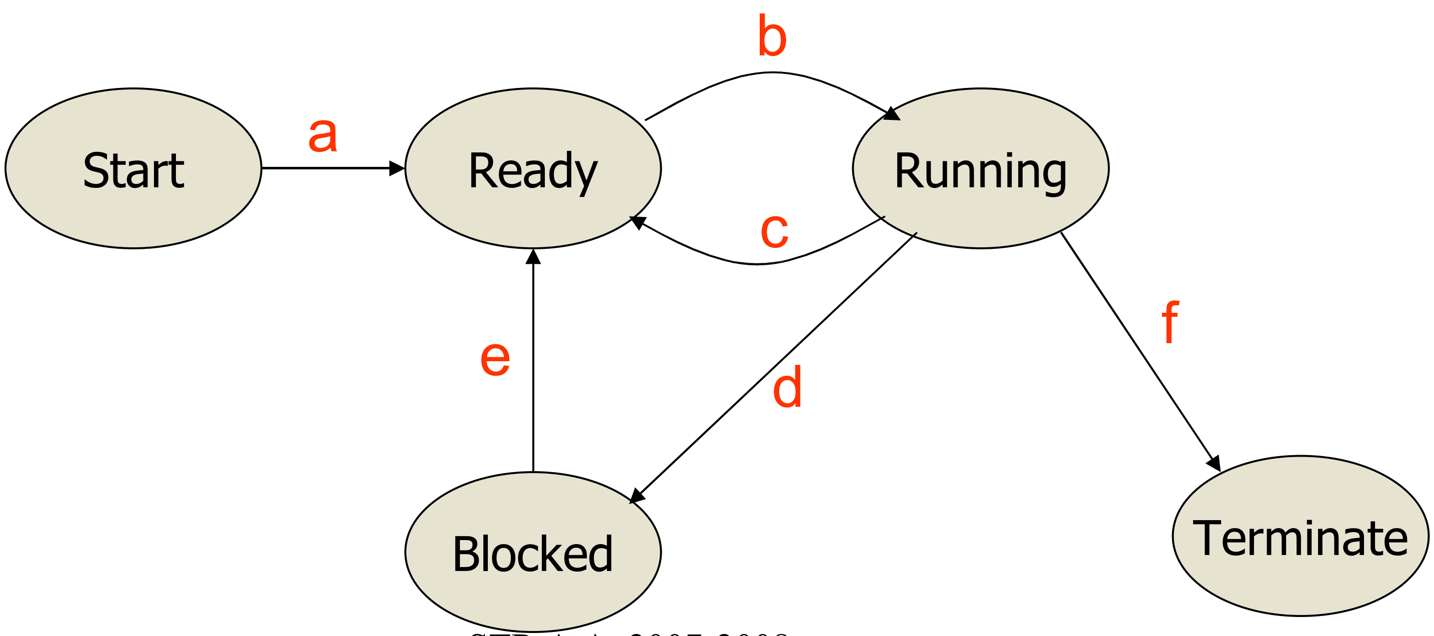
- Every time the program is launched, a new process is generated and run

Process states

- The OS can execute many processes “at the same time”
  - Only one process can execute at some instant of time, however, by using *time sharing*, the OS can switch between all the processes that are *ready* to execute
  - From the OS point of view, it is important to understand which process is *ready* to execute
- Each process, during its lifetime can be in one of the following states
  - Starting (the process is being created)
  - Ready (the process is ready to be executed)
  - Executing (the process is executing)
  - Blocked (the process is waiting on a condition)
  - Terminating (the process is about to terminate)

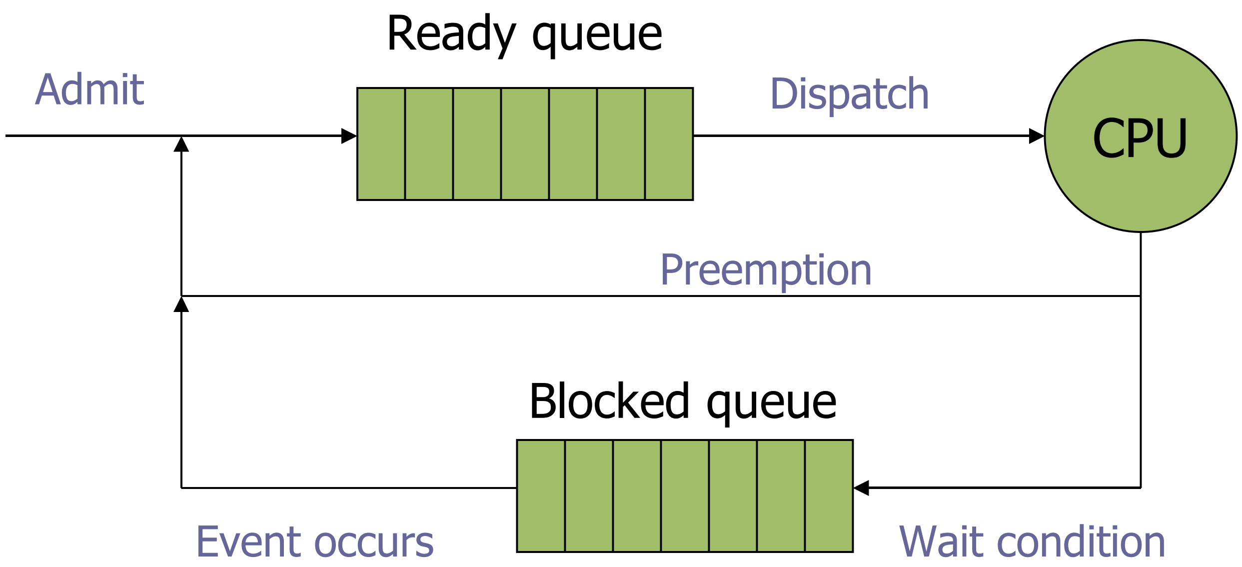
Process states

a) Creation — The process is created
b) Dispatch — The process is selected to execute
c) Preemption — The process leaves the processor
d) Wait on condition — The process is blocked on a condition
e) Condition true — The process is unblocked
f) Exit — The process terminates



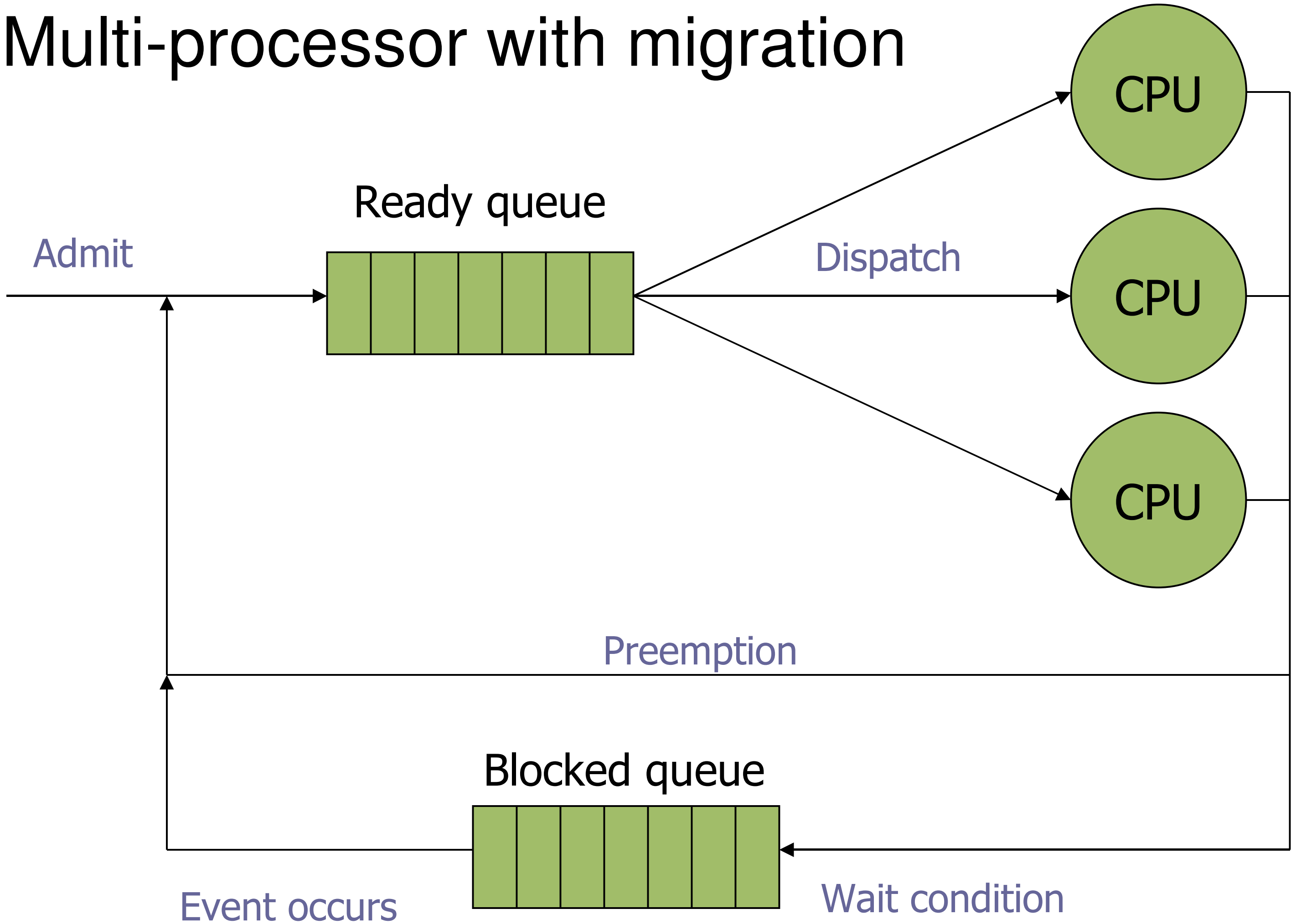
Process queues

- Single processor

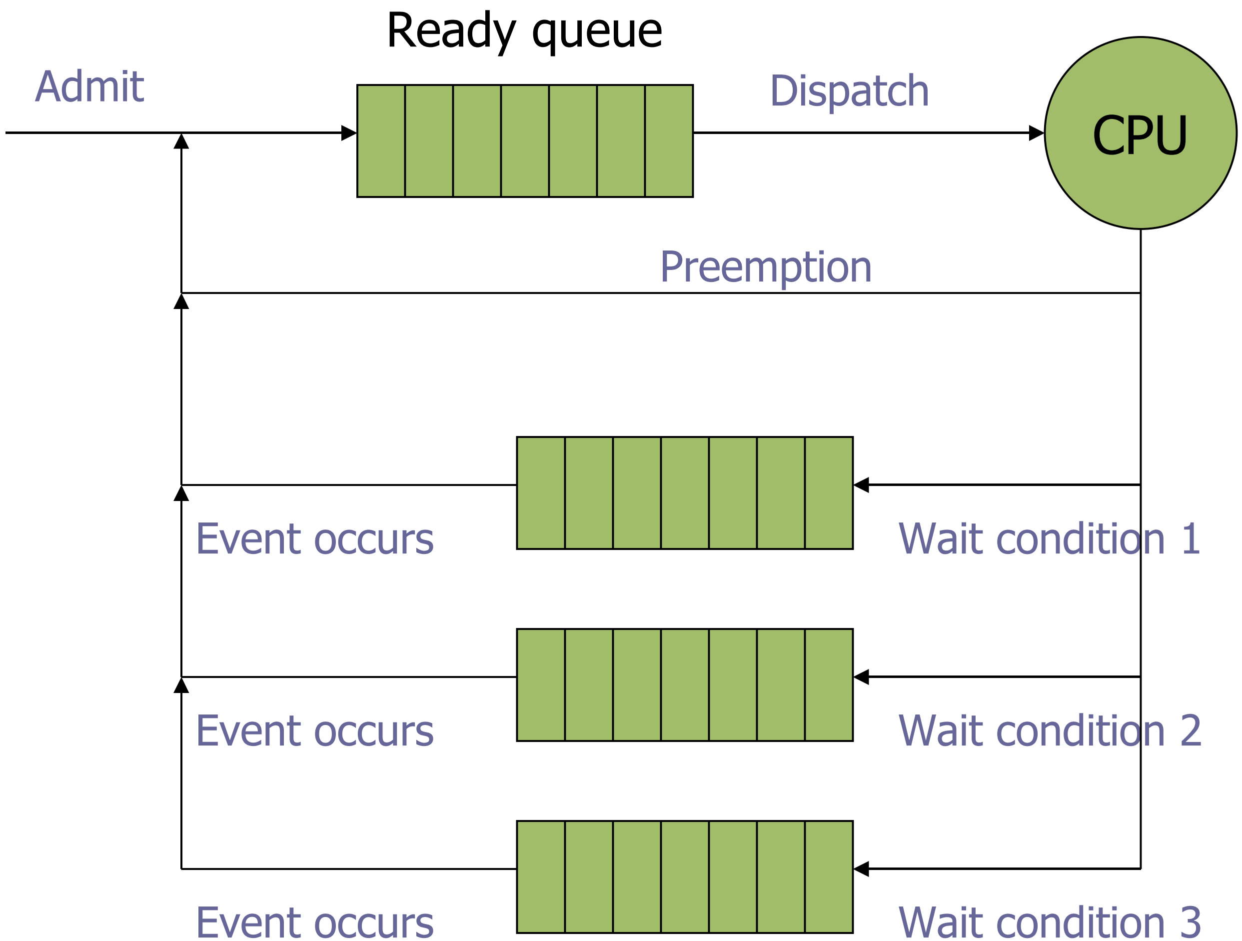


Process queues

- Multi-processor with migration

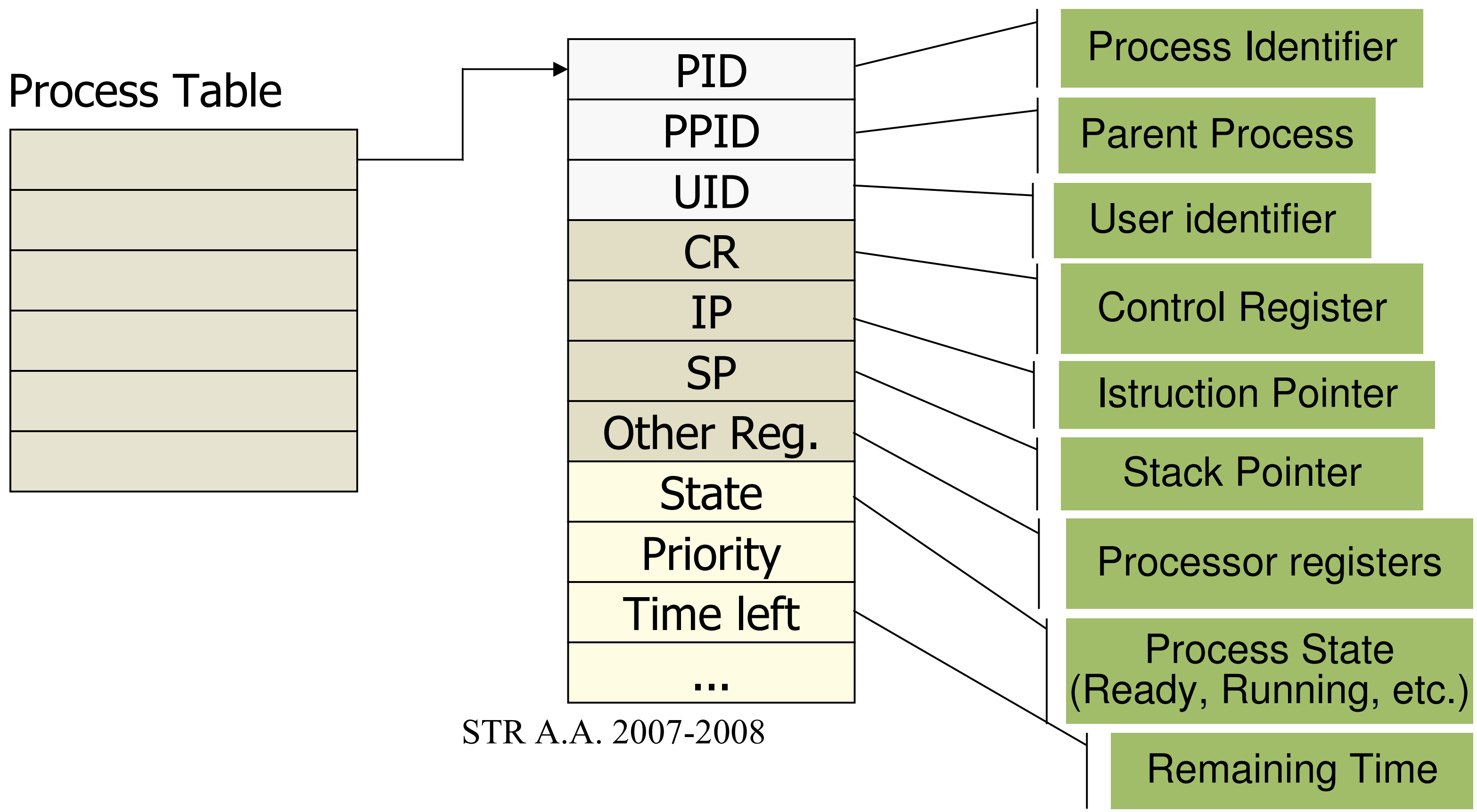


Multiple blocking queues



Process Control Block

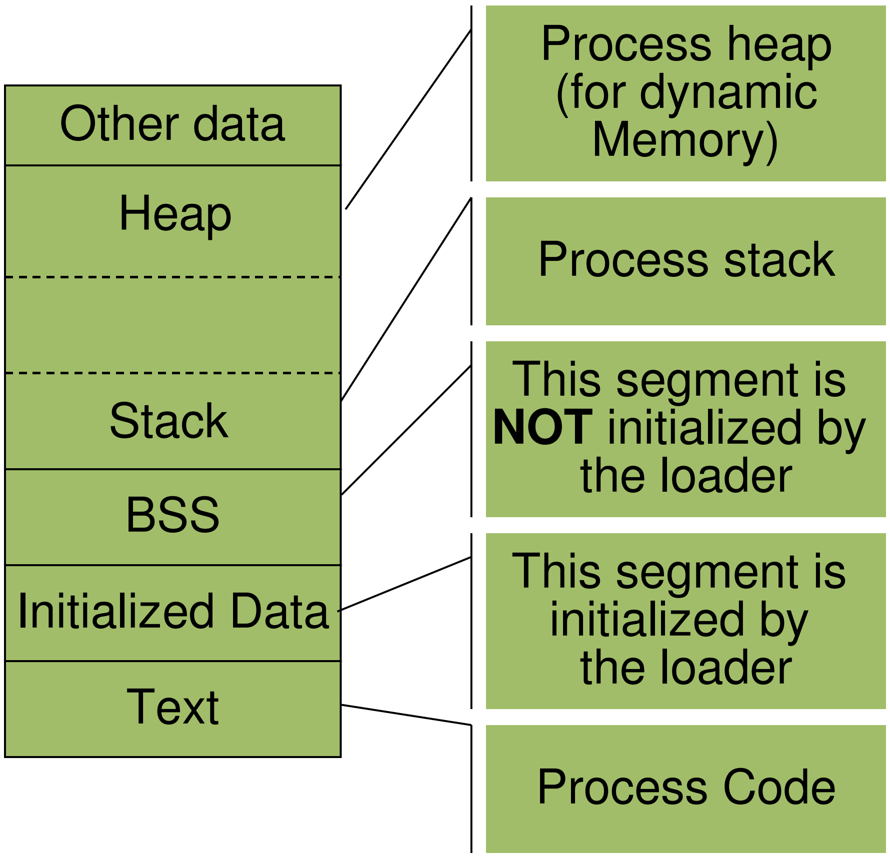
- It contains all the data concerning on process
- All PCBs are stored in the Process Table



The role of PCB

- Virtually every routine in the OS uses the PCBs
  - The scheduler
  - The Virtual memory
  - The Virtual File System
  - Interrupt handlers (I/O devices)
  - ...
- It can only be accessed by the OS
- The user can access some of the information in the PCB by using appropriate system calls
- The PCB is a critical point of any OS

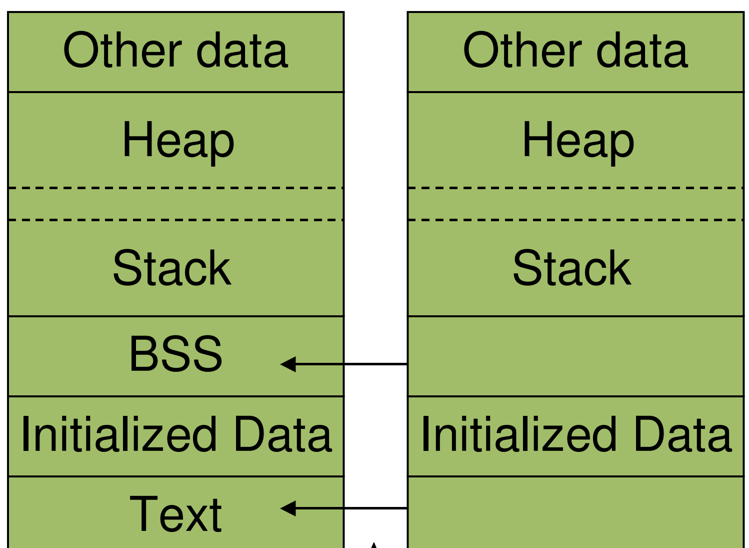
Memory layout of a Process



- In some system, the PCB is in the “other data” section

Memory protection

- Every process has its own memory space
  - Part of it is “private to the process”
  - Part of it can be shared with other processes
  - For examples: two processes that are instances of the same program will probably share the TEXT part
  - If two processes want to communicate by shared memory, they can share a portion of the data segment



Memory Protection

- By default, two processes cannot share their memory
  - If one process tries to access a memory location outside its space, a processor exception is raised (trap) and the process is terminated
  - The famous “Segmentation Fault” error!!
  - We will come back to this when we study the Virtual Memory

Modes of operation (revised)

- Every modern processor supports at least 2 modes of operation
  - User
  - Supervisor
  - The Control Register (CR) contains one bit that tells us in which mode the processor is running
- Operating system routines run in supervisor mode
  - They need to operate freely on every part of the hardware with no restriction
  - User code runs into user mode
- Mode switch
  - Every time we go from user to supervisor mode or viceversa

Mode switch

- It can happen in one of the following cases
  - Interrupts or traps
    - In this case, before calling the interrupt handler, the processor goes in supervisor mode and disables interrupts
    - Traps are interrupts that are raised when a critical error occurs (for example, division by zero, or page fault)
    - Returning from the interrupt restores the previous mode
  - Invoking a special instruction
    - In the Intel family, it is the INT instruction
    - This instruction is similar to an interrupt
    - It takes a number that identifies a “service”
    - All OS calls are invoked by calling INT
    - Returning from the handler restores the previous mode

Example of system call

```
int fd,n;
char buff[100];

fd = open("Dummy.txt", O_RDONLY);
n =  read(fd, buff, 100);
```

- Saves parameters on the stack
- Executes INT 20h
  1. Change to supervisor mode
  2. Save context
  3. Execute the function open
  4. Restores the context
  5. IRET
- Back to user mode
- Delete parameters from the stack

- The "open" system call can potentially block the process!
- In that case we have a "process switch"

Process switch

- It happens when
  - The process has been “preempted” by another higher priority process
  - The process blocks on some condition
  - In time-sharing systems, the process has completed its “round” and it is the turn of some other process
- We must be able to restore the process later
  - Therefore we must save its state before switching to another process

The “exec” pointer

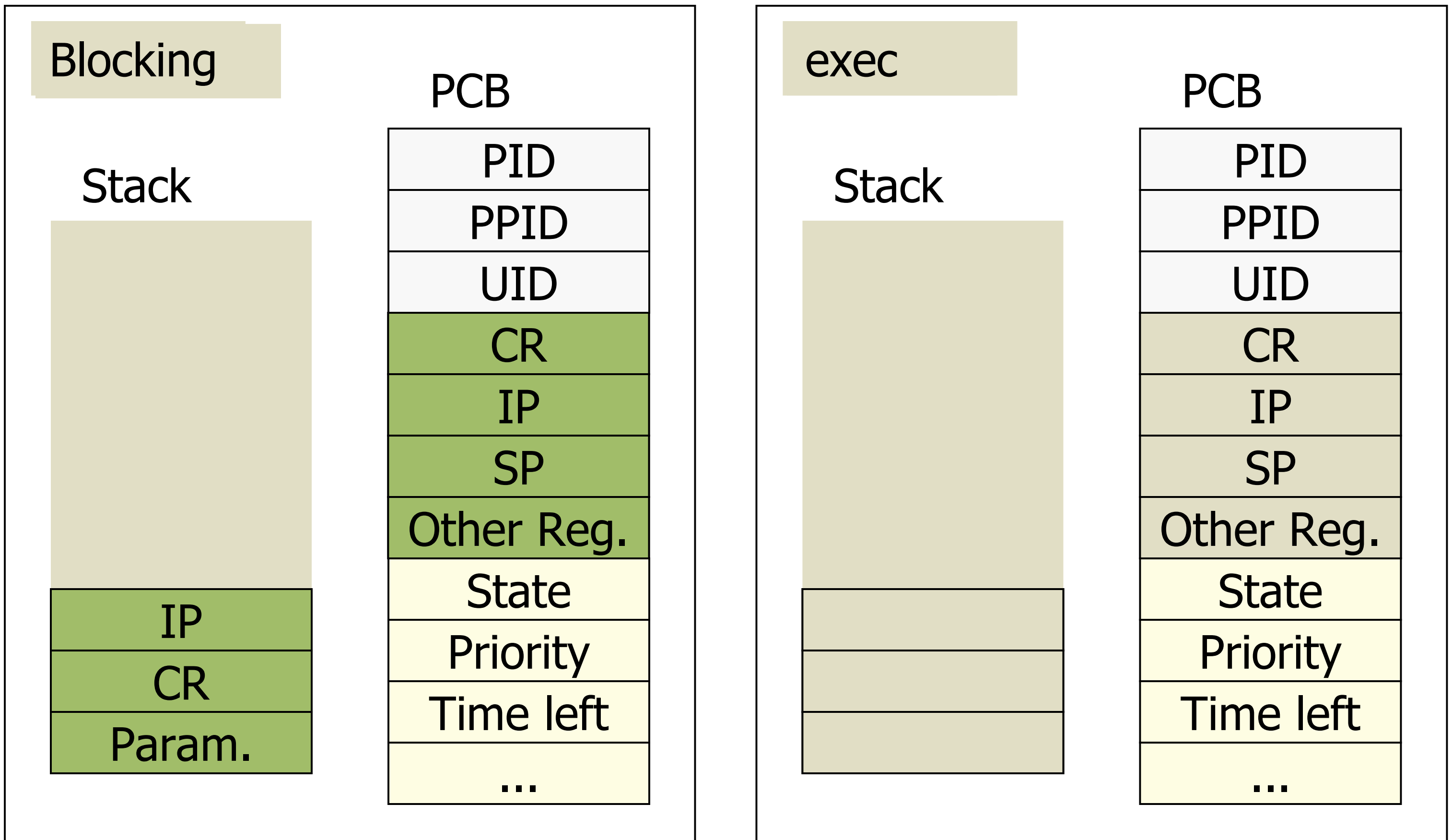
- Every OS has one pointer (“exec”) to the PCB of the running process
  - The status of the “exec” process is RUNNING
- When a process switch occurs,
  - The status of the “exec” process is changed to BLOCKING or READY
  - The scheduler is called
  - The scheduler selects another “exec” from the ready queue

System call with process switch

- Saves parameters on stack
- INT 20h
  - Change to supervisor mode
  - Save context in the PCB of “exec” (including SP)
  - Execute the code
    - The process change status and goes into BLOCKING mode
  - Calls the scheduler
    - Moves “exec” into the blocking queue
    - Selects another process to go into RUNNING mode
    - Now exec points to the new process
  - Restores the context of “exec” (including SP)
    - This changes the stack
  - IRET
    - Returns to where the new process was interrupted

Stacks

- Executing IRET

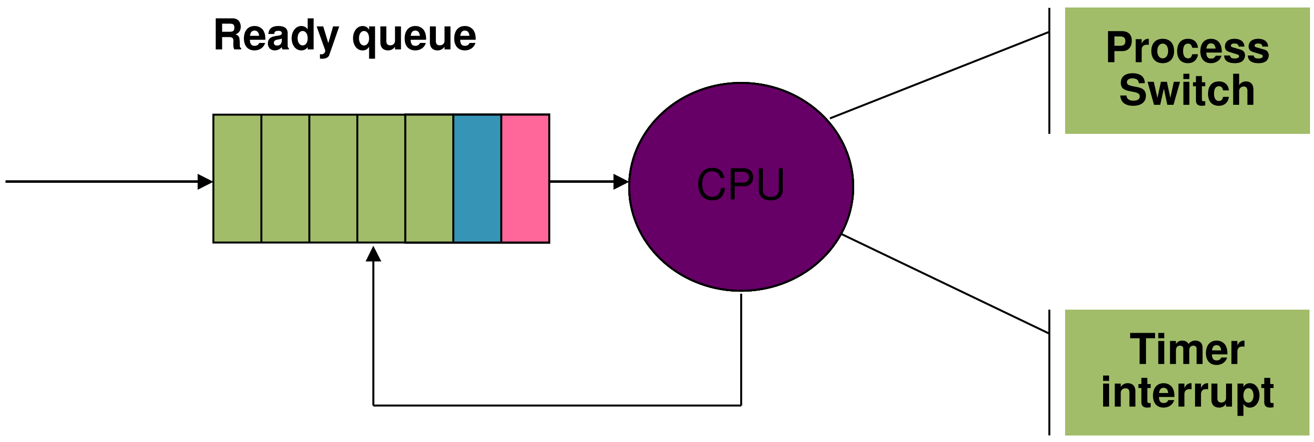


Process switch

- This is only an example
  - Every OS has little variations on the same theme
  - For example, in most cases, registers are saved on the stack, not on the PCB
- You can try to look into some real OS
  - Linux
  - Free BSD
  - Shark (http://shark.sssup.it)
  - Every OS is different!

Time sharing systems

- In time sharing systems,
  - Every process can execute for maximum one round
    - For example, 10msec
  - At the end of the round, the processor is given to another process



Interrupt with process switch

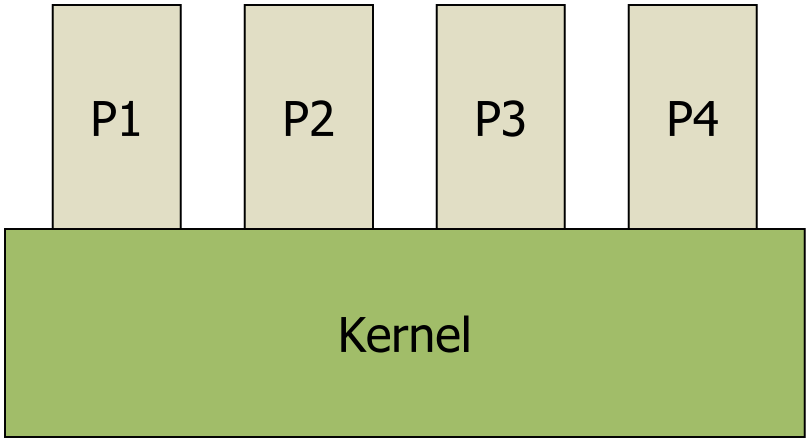
- It is very similar to the INT with process switch
  - An interrupt arrives
  - Change to supervisor mode
  - Save CR and IP
  - Save processor context
  - Execute the interrupt handler
  - Call the scheduler
    - This may change the “exec” pointer
  - IRET

Causes for a process switch

- A process switch can be
  - Voluntary: the process calls a blocking primitive, i.e. it executes an INT
    - For example, by calling a read() on a blocking device
  - Non-voluntary: an interrupt arrives that causes the process switch
    - It can be the timer interrupt , in time-sharing systems
    - It can be an I/O device which unblocks a blocked process with a higher priority

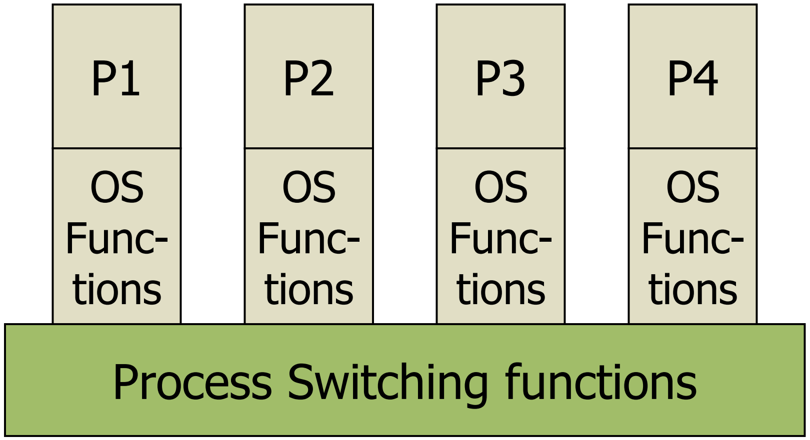
OS structures

- Non process kernel
  - The kernel executes in its own context
  - Every time we do a mode switch,
    - Registers are saved
    - The stack is changed



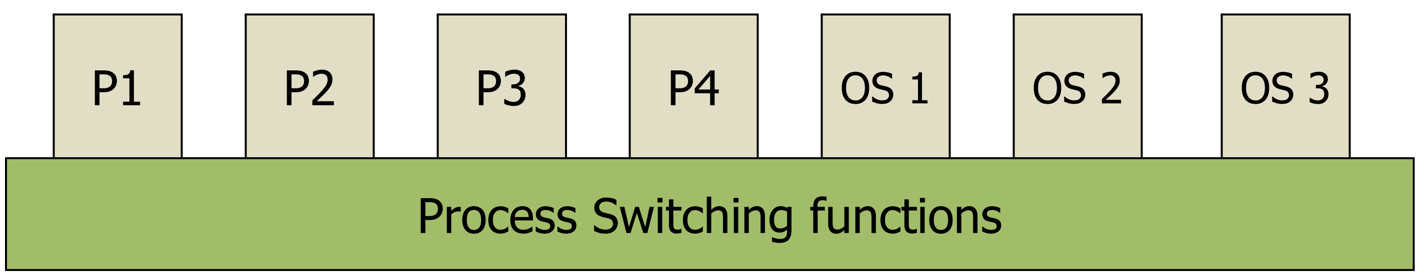
Execution within User processes

- The kernel routines execute in the context of the running process
  - The stack is not changed
  - Local variables of the kernel are saved on top of the process stack
  - Only a small part (the process switch) is independent of the process
- Linux and Shark and many others have this structure



Process-based OS

- This is the typical microkernel structure
  - Each routine (service) is a separate process
  - Every system call involves a process switch
- Advantages:
  - Modularity, Composability, Extendibility
- Disadvantages
  - Overhead



Processes and Threads

Processes

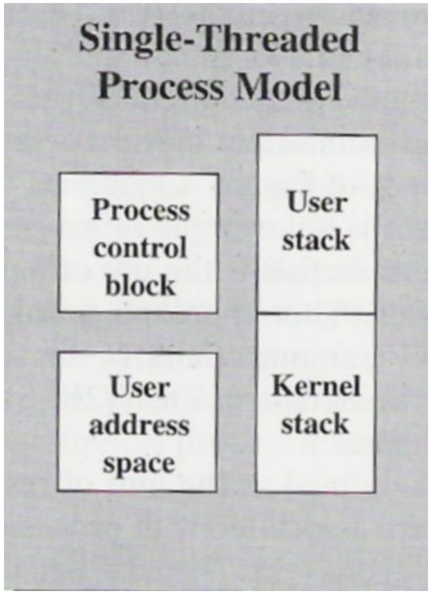
- We can distinguish two aspects in a process
- Resource Ownership
  - A process includes a virtual address space, a process image (code + data)
  - It is allocated a set of resources, like file descriptors, I/O channels, etc
- Scheduling/Execution
  - The execution of a process follows an ececution path, and generates a trace (sequence of internal states)
  - It has a state (ready, Running, etc.)
  - And scheduling parameters (priority, time left in the round, etc.)

Multi-threading

- Many OS separate these aspects, by providing the concept of thread
- The process is the “resource owner”
- The thread is the “scheduling entity”
  - One process can consists of one or more *threads*
  - Threads are sometime called (improperly) lightweight processes
  - Therefore, on process can have many different (and concurrent) traces of execution!

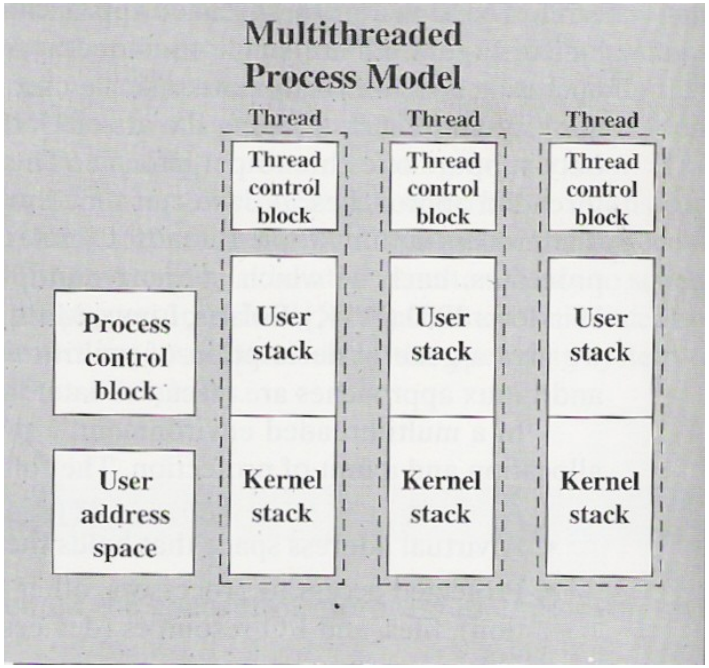
Single threaded Process Model

- In the single-threaded process model one process has only one thread
  - One address space
  - One stack
  - One PCB only



Multi-threaded process model

- In the multi-threaded process model each process can have many threads
  - One address space
  - One PCB
  - Many stacks
  - Many TCB (Thread Control blocks)
  - The threads are scheduled directly by the global scheduler



Threads

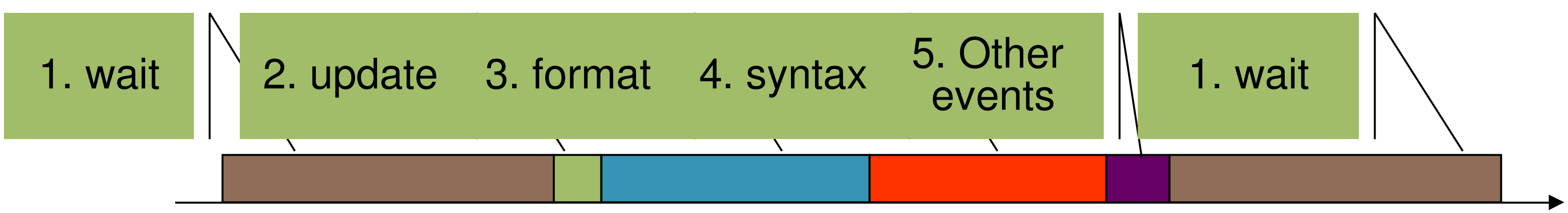
- Generally, processes do not share memory
  - To communicate between process, it is necessary to user OS primitives
  - Process switch is more complex because we have to change address space
- Two threads in the same process share the same address space
  - They can access the same variables in memory
  - Communication between threads is simpler
  - Thread switch has less overhead

Processes vs. Threads

- Processes are mainly used to compete for some resource
  - For example, two different users run two separate applications that need to print a file
  - The printer is a shared resource, the two processes compete for the printer
- Threads are mainly used to collaborate to some goal
  - For example, one complex calculation can be split in two parallel phases, each thread does one phase
  - In a multi-processor machine the two threads go in parallel and the calculation becomes faster

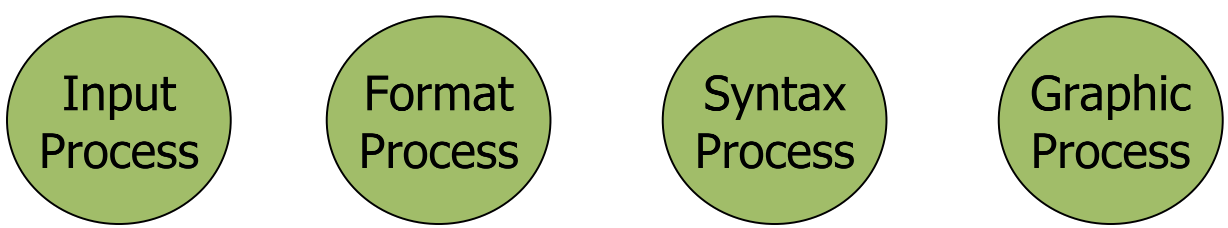
Example - I

- Consider a Word Processor application
- Main cycle
  1. Wait for input from the keyboard
  2. Update the document
  3. Format the document
  4. Check for syntax errors
  5. Check for other events (i.e. temporary save)
  6. Return to 1
- One single process would be a waste of time!



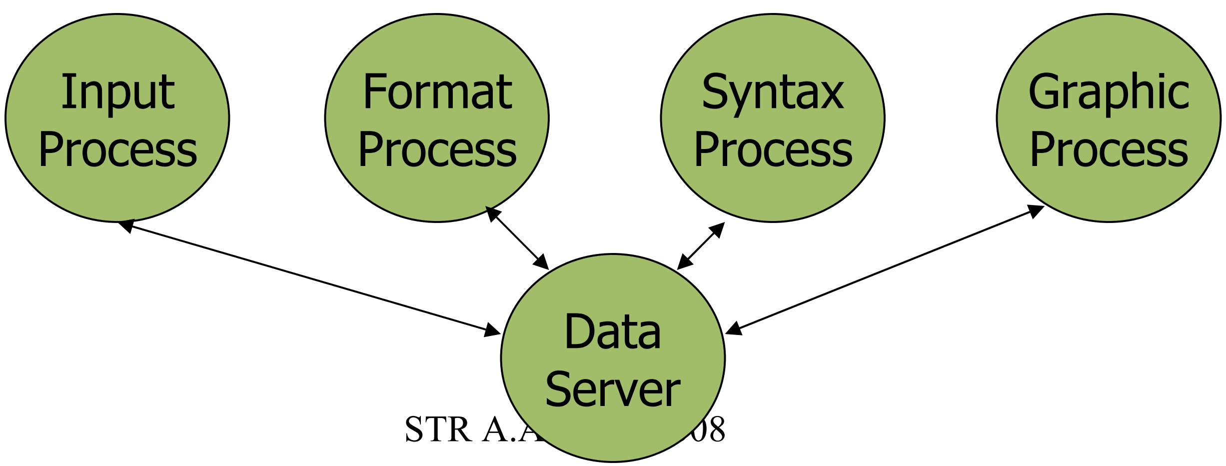
Example - II

- Problems
  - Most of the time, the program waits for input
    - Idea, while waiting we could perform some other task
  - Activities 3 and 4 (formatting and syntax checking) are very time consuming
    - Idea: let's do them while waiting for input
- Solution with multiple processes
  - One process waits for input
  - Another process periodically formats the document
  - A third process periodically performs a syntax checking
  - A fourth process visualize the document



Example - III

- Problem with multiple processes
  - All processes needs to access the same data structure, the document
  - Which process holds the data structure?
  - Solution 1: message passing
    - A dedicated process holds the data, all the others communicate with it to read/update the data
    - Very inefficient!

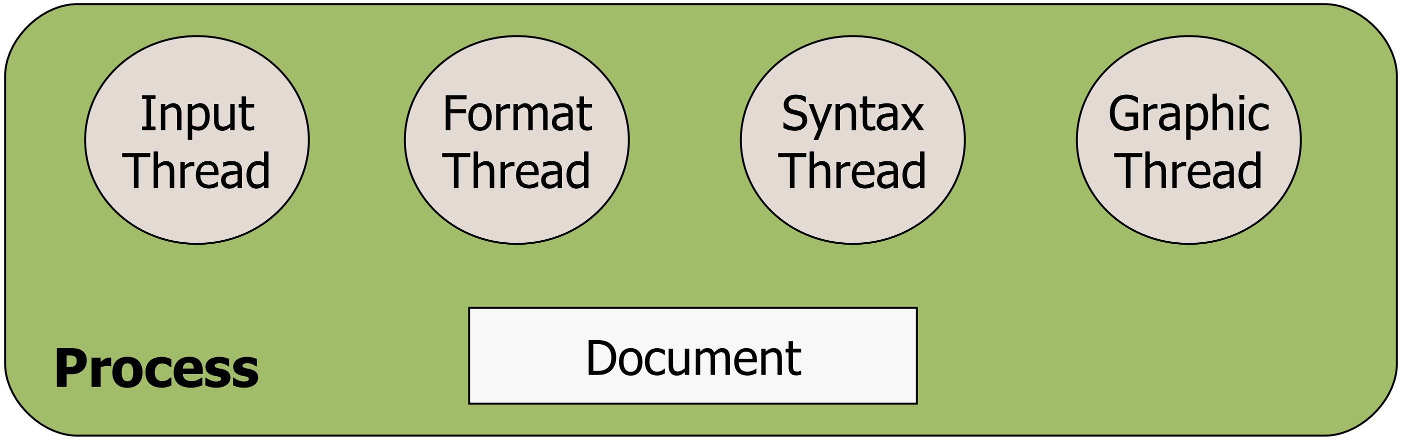


Example - IV

- Another solution...
  - Solution 2: shared memory
    - One process holds the data and makes that part of its memory shareable with the others
  - Still not very efficient:
    - We have a lot of process switches
    - Memory handling becomes very complex

Why using threads

- Speed of creation
  - Creating a thread takes far less time than creating a process
- Speed of switching
  - Thread switch is faster than process switch
- Shared memory
  - Threads of the same process run in the same memory space
  - They can naturally access the same data!



Threads support in OS

- Different OS implement threads in different ways
  - Some OS supports directly only processes
    - Threads are implemented as "special processes"
  - Some OS supports only threads
    - Processes are threads' groups
  - Some OS natively supports both concepts
    - For example Windows NT
- We will come back to this part later, after we have studied the problem of synchronization
- By now we need to abstract away the different implementations

Summary

- Important concepts
  - Process: provides the abstraction of memory space
  - Threads: provide the abstraction of execution trace
  - The scheduler manages threads!
- Processes do not normally share memory
- Two threads of the same process share memory
- We need to explore all the different ways in which two threads can communicate
  - Shared memory
  - Message passing
- In the next section we will only refer to threads

Esercizio

- Considerate il seguente task

```
void *threadA(void *arg)
{
  int i;
  double s = *((double *) arg);
    double vect[3];
    for (i=0; i<3; i++) vect[i] = 0;
    while (1) {
    multiply(vect);
    if (length(vect) >s) normalize(vect);
    task_endcycle();
    }
}
```

Ipotesi di lavoro

- processore a 32 bit
- Int = 4 bytes
- Double = 8 bytes
- Char = 1 byte

```
double mat[3][3];

void multiply(double v[])
{
    int i,j;
    double ris[3];
    for(i=0; i<3; i++) {
        ris[i] = 0;
        for (j=0; j<3; j++) ris[i] += mat[i][j] * v[i];
    }
    for (i=0; i<3; i++) v[i] = ris[i];
    return;
}

double length(double v[])
{
   return sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
}

void normalize(double v[])
{
    int i;
    double l = length(v);
    for (i=0; i<3; i++) v[i] /= l;
    return;
}
```

Esercizio

- Domande:
  - Disegnare la struttura dello stack e calcolare la sua dimensione in byte
  - Descrivere cosa succede quando arriva una interruzione
  - In un sistema time sharing, descrivere cosa succede quando il quanto di esecuzione del thread è terminato