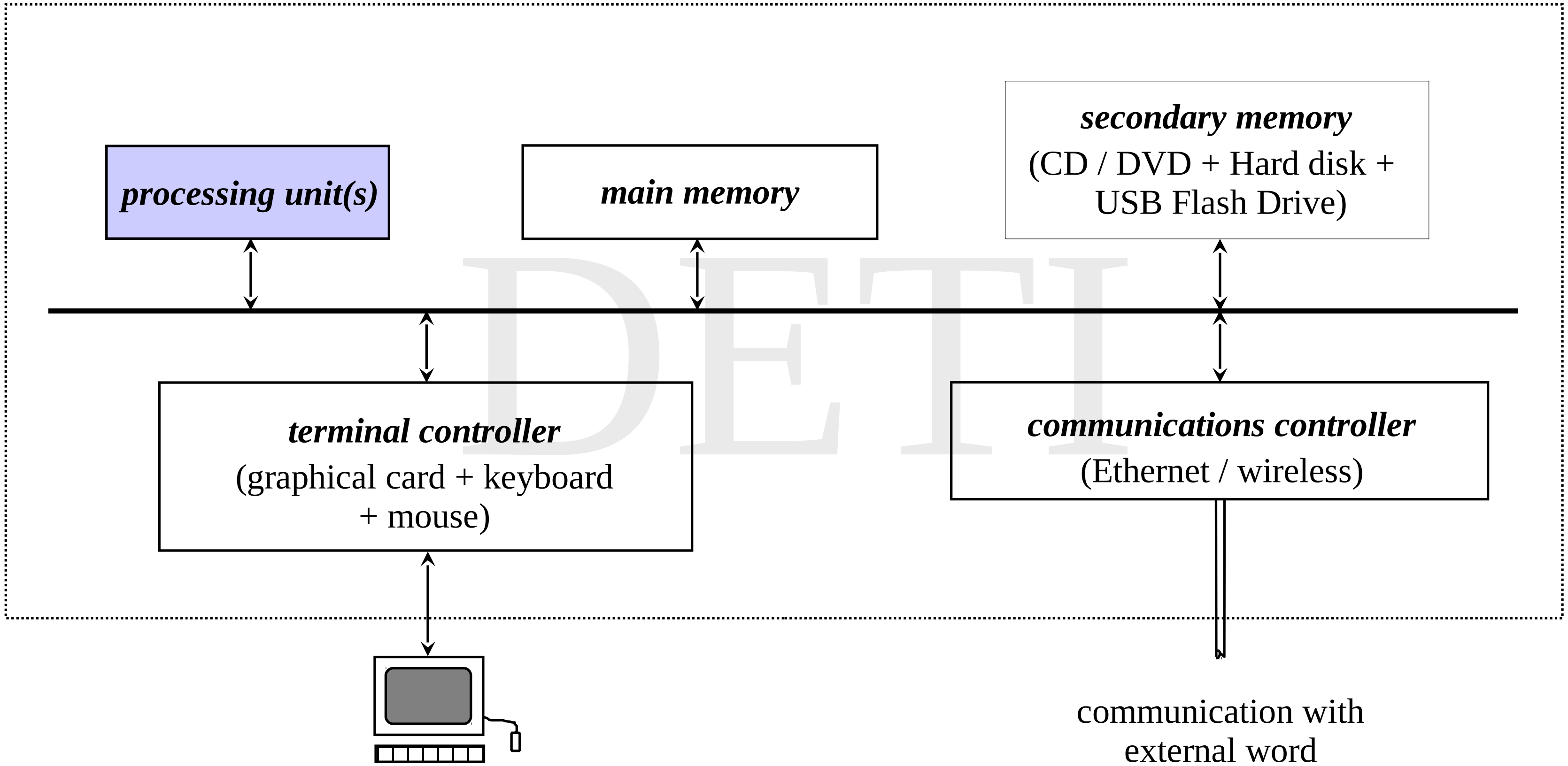
Fundamentos de Sistemas Operativos / Sistemas de Operação

Processes and threads

Artur Pereira / António Rui Borges

Typical computational system



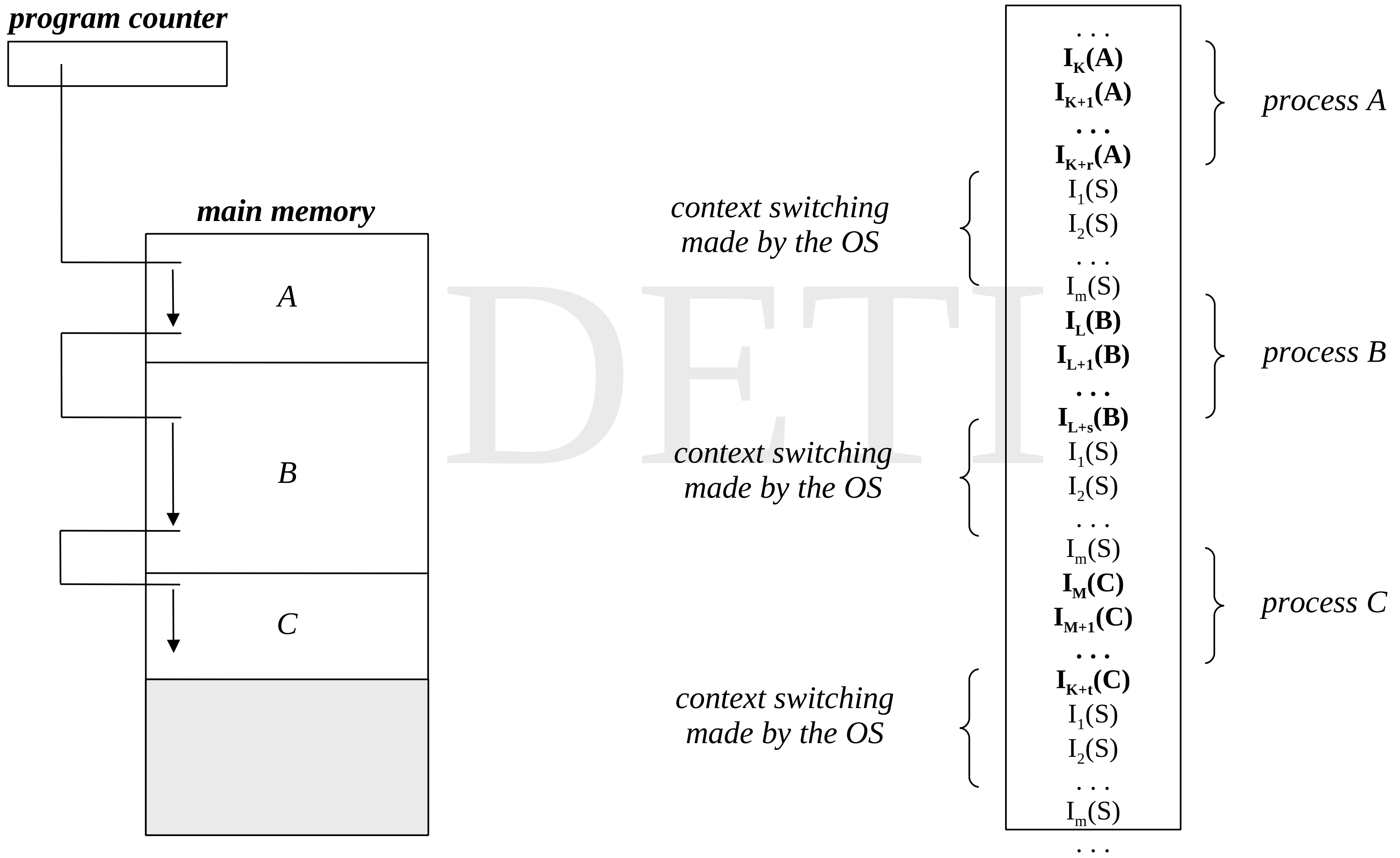
Contents

- *Program vs. Process*
- *Process model*
- *State diagram of a process*
- *Process creation in Unix*
- *Address space of a Unix program*
- *Threads and multithreading*
- *Threads in Linux*
- *Process switching*
- *Bibliography*

Program vs. Process

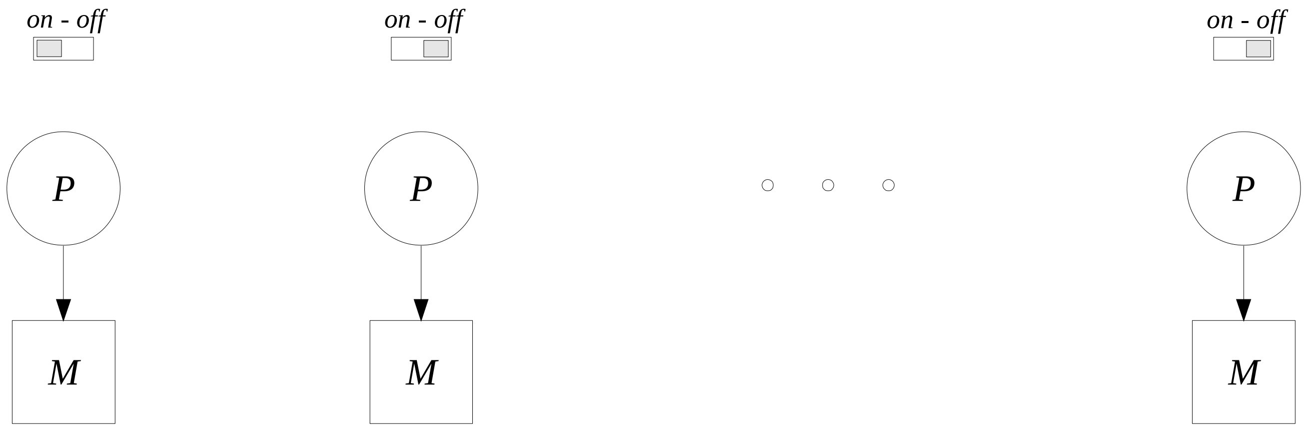
- Program – set of instructions describing how a task is performed by a computer
  - In order for the task to be actually performed, the corresponding program has to be executed
- Process – an entity that represents a computer program being executed
  - it represents an activity of some kind
  - it is characterized by:
    - code and data (actual values of the diferent variables) of the associated program (*addressing space*);
    - actual values of the processor internal registers
    - input and output data (data that are being transfered from input devices and to output devices)
    - state of execution
- Different processes can be running the same program
- In general, there are more processes than processors – *multiprogramming*

Execution in a multiprogrammed environment



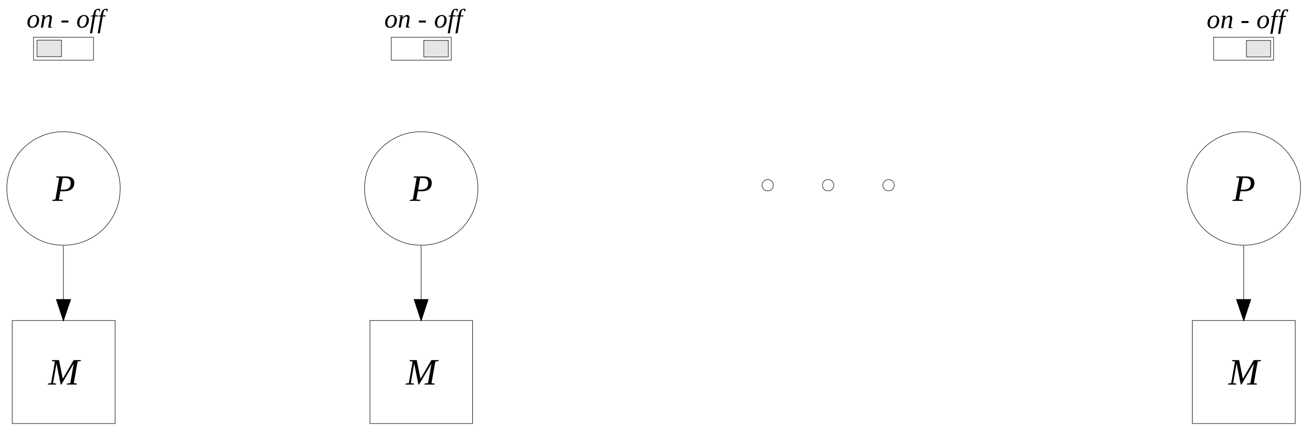
Process model

- In multiprogramming, the activity of the processor, because it is switching back and forth from process to process, is hard to perceive
- Thus, it is better to assume the existence of a number of virtual processors, one per existing process
  - Turning *off* one virtual processor and *on* another, corresponds to a process switching
  - number of active virtual processors <= number of real processors



Process model

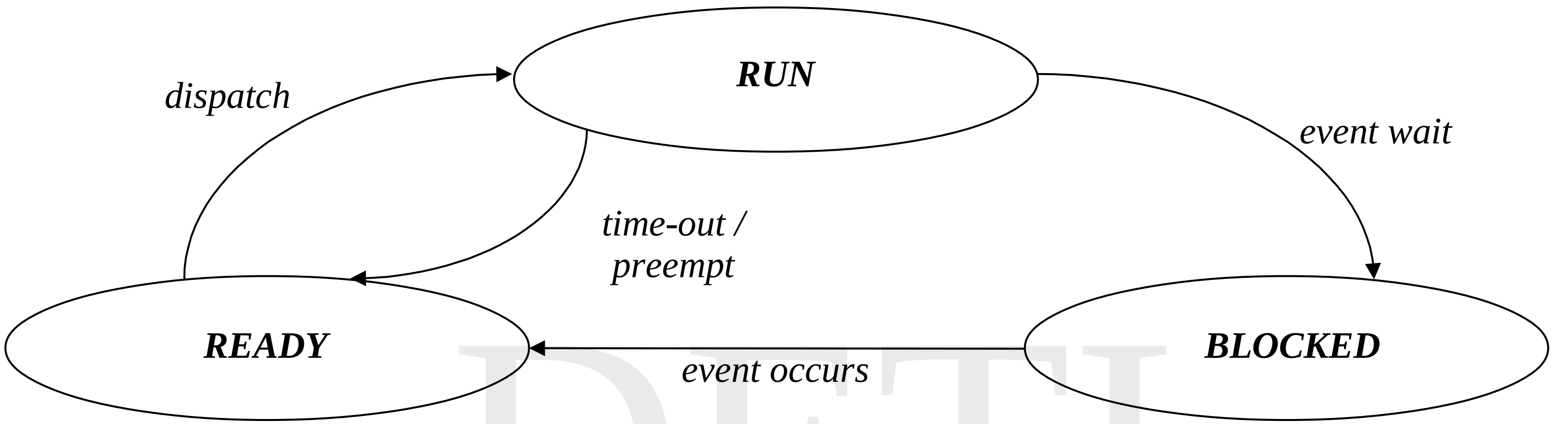
- The switching between processes, and thus the switching between virtual processors, can occur for different reasons, possible not controlled by the running program
- Thus, to be viable, this process model requires that
  - *the execution of any process is not affected by the instant in time or the location in the code where the switching takes place*
  - *no restrictions are imposed on the total or partial execution times of any process*



State diagram of a process

- During its existence, a process can be in different situations, named *states*
  - The most important are:
    - *run* – the process is in possession of a processor, and thus running
    - *blocked* – the process is waiting for the occurrence of an external event (access to a resource, end of an input/output operation, etc.)
    - *ready* – the process is ready to run, but waiting for the availability of a processor to start/resume its execution
- Transitions between states usually result from external intervention, but can in some cases be triggered by the process itself
- The part of the operating system that handles these transitions is called the (*processor*) *scheduler*, and is an integral part of its kernel
  - Different policies exist to control the firing of these transitions

State diagram of a process

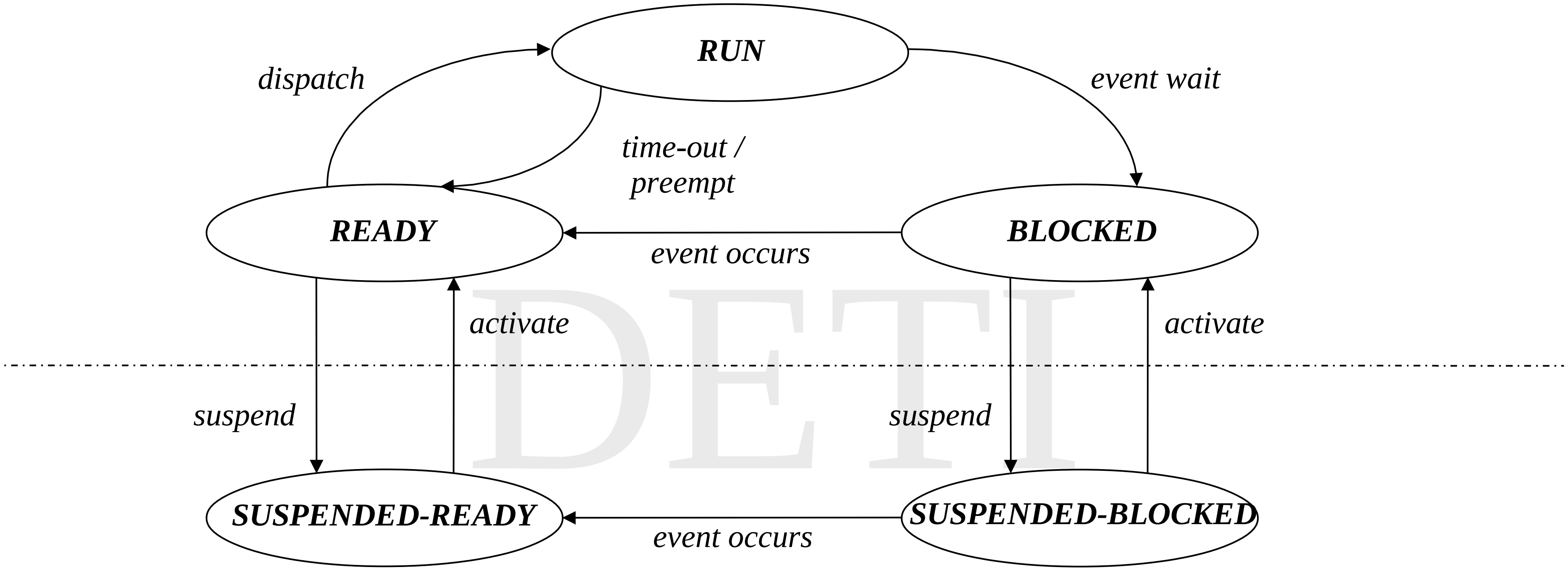


- *dispatch* – one of the processes ready to run is selected and is given the processor
- *event wait* – the running process is prevented to proceed, awaiting the occurrence of an external event
- *event occurs* – the external event occurred and the process must now wait for the processor
- *time-out* – the time slot assigned to the running process get to the end, so the process is removed from the processor
- *preempt* – a higher priority process get ready to run, so the running process is removed from the processor

State diagram of a process

- The main memory is finite, which limits the number of coexisting processes.
- A way to overcome this limitation is to use an area in secondary memory to extend the main memory
  - This is called *swap area* (can be a disk partition or a file)
  - A non running process, or part of it, can be *swapped out*, in order to free main memory for other processes
  - That process will be later on *swapped in*, after main memory becomes available
- Two new states should be added to the process state diagram to incorporate these situations:
  - *suspended-ready* – the process is *ready* but swapped out
  - *suspended-blocked* – the process is *blocked* and swapped out

State diagram of a process

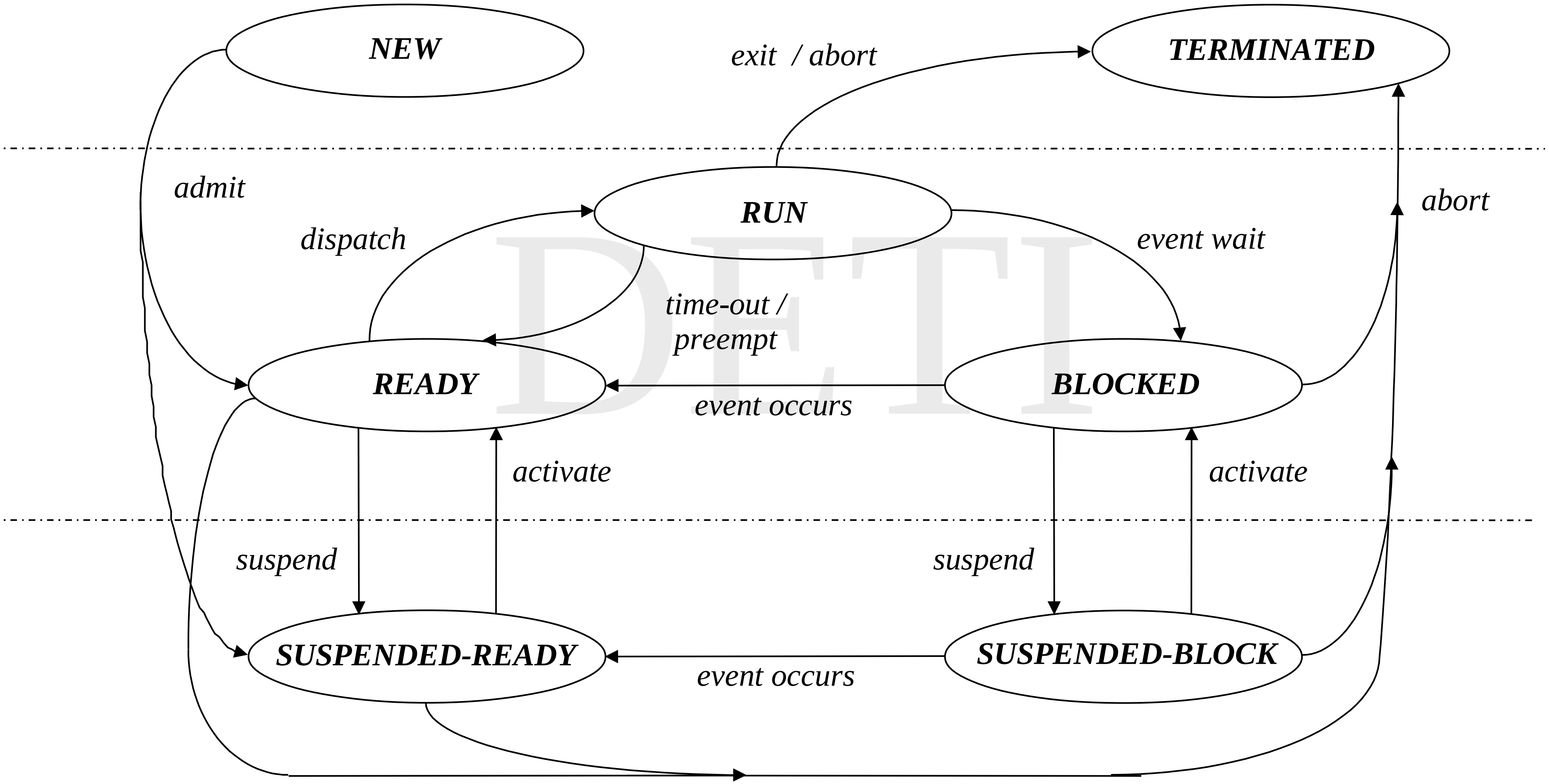


- Two new transitions appear:
  - *suspend* – the process is *swapped out*
  - *activate* – the process is *swapped in*

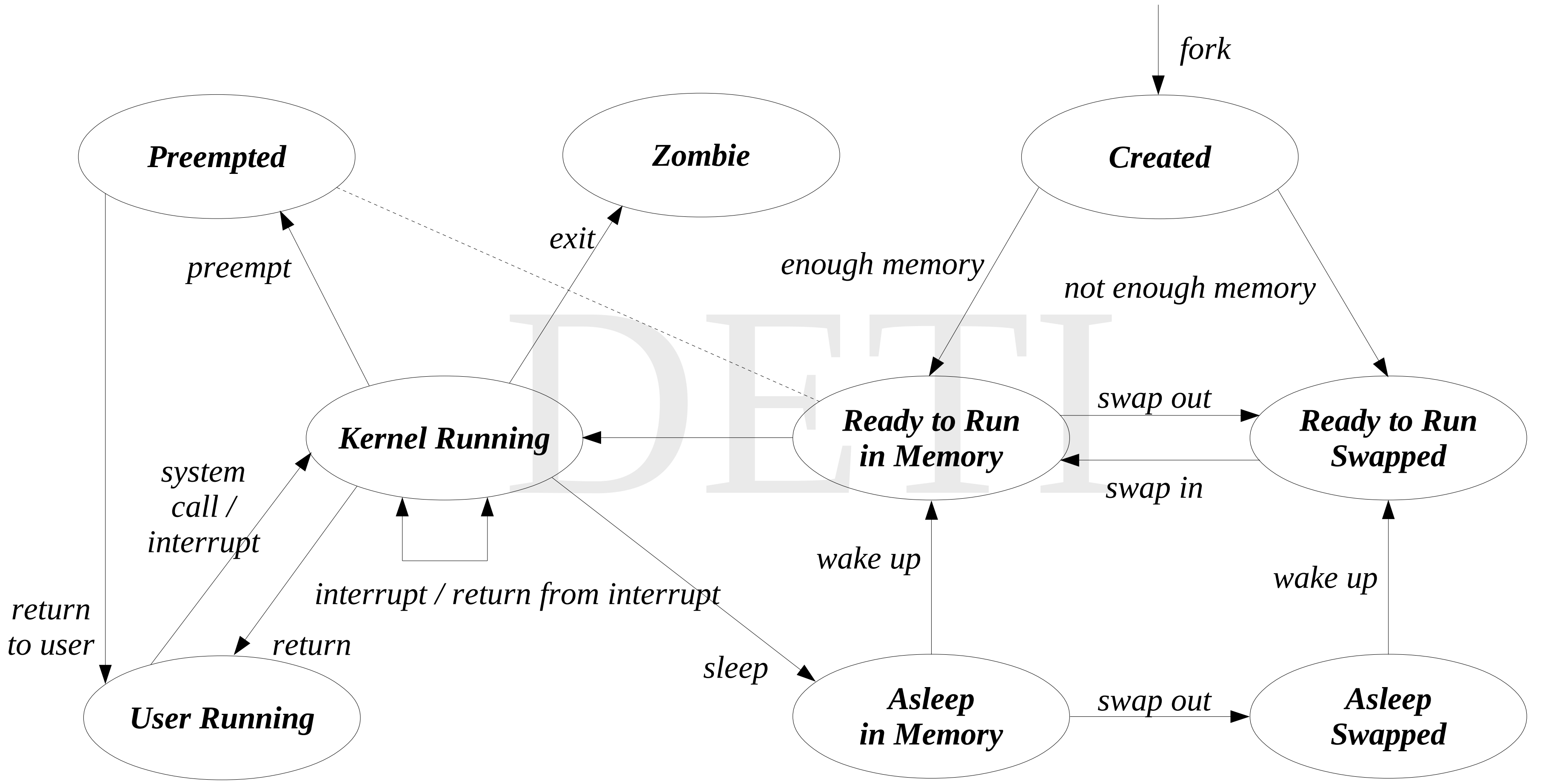
State diagram of a process

- The previous state diagram assumes processes are timeless
  - Apart from some system processes this is not true
  - Processes are created, exist for some time, and eventually finish
- Two new states are required to represent creation and termination
  - *new* – the process has been created but not yet admitted to the pool of executable processes (the process data structure is been initialized)
  - *terminated* – the process has been released from the pool of executable processes (some actions are still required before the process is discarded)
- three new transitions exist
  - *admit* – the process is admitted (by the OS) to the pool of executable processes
  - *exit* – the running process indicates the OS it has completed
  - *abort* – the process is forced to terminate (because of a fatal error or because an authorized process aborts its execution)

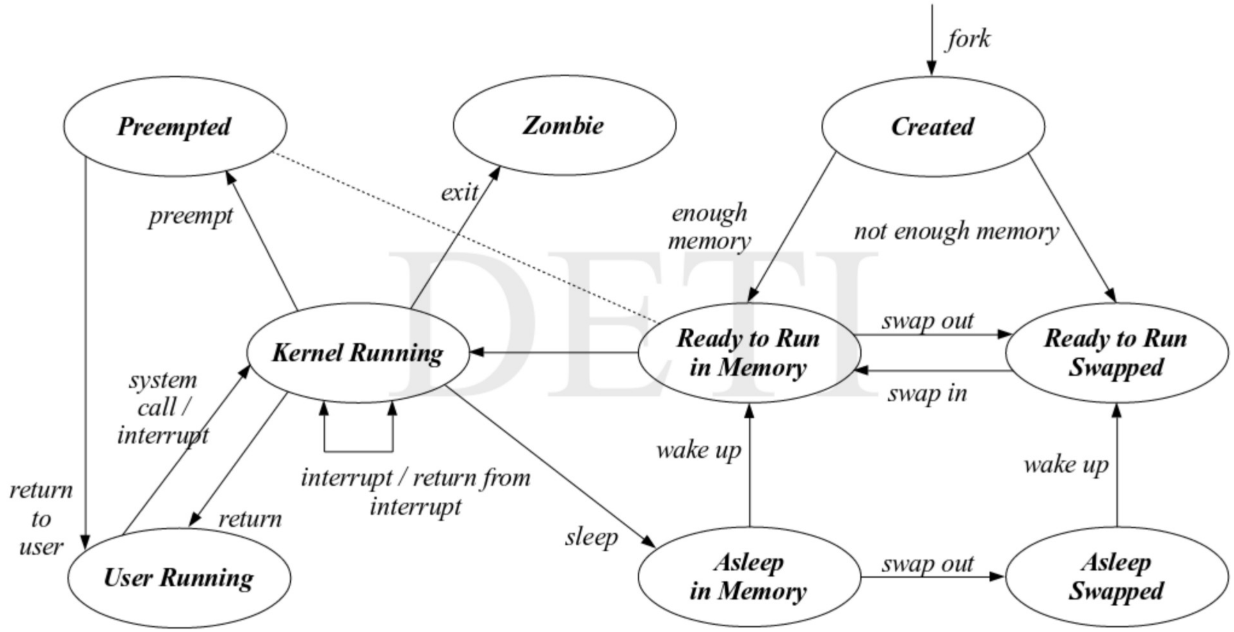
State diagram of a process



State diagram of a Unix process

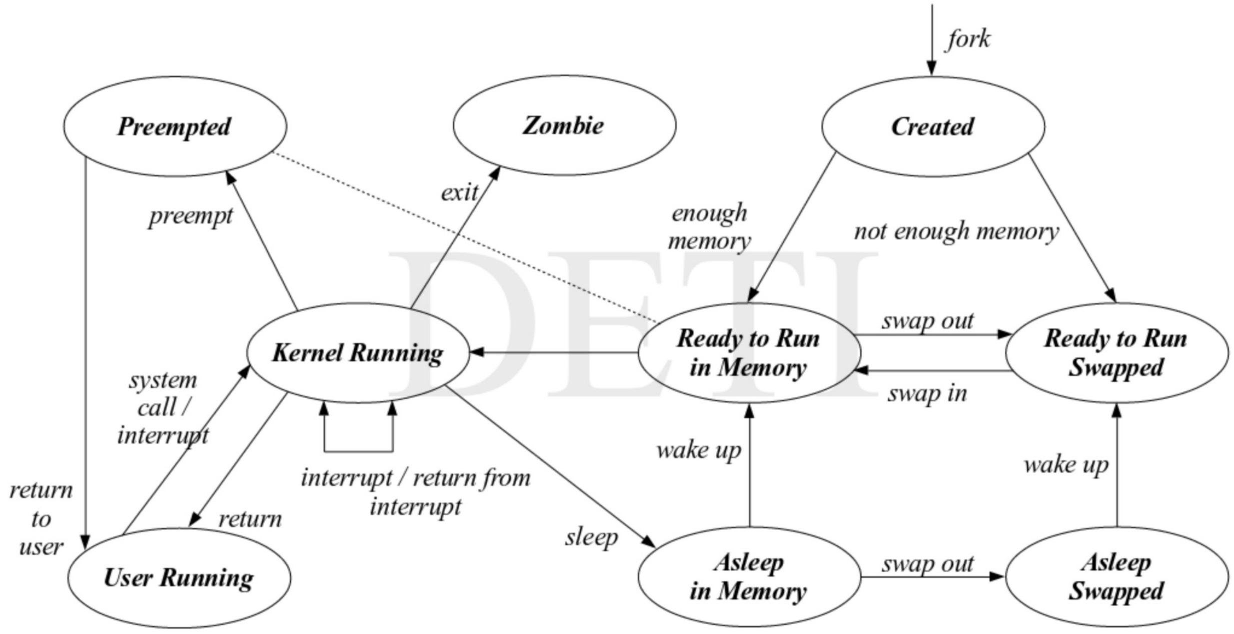


State diagram of a Unix process



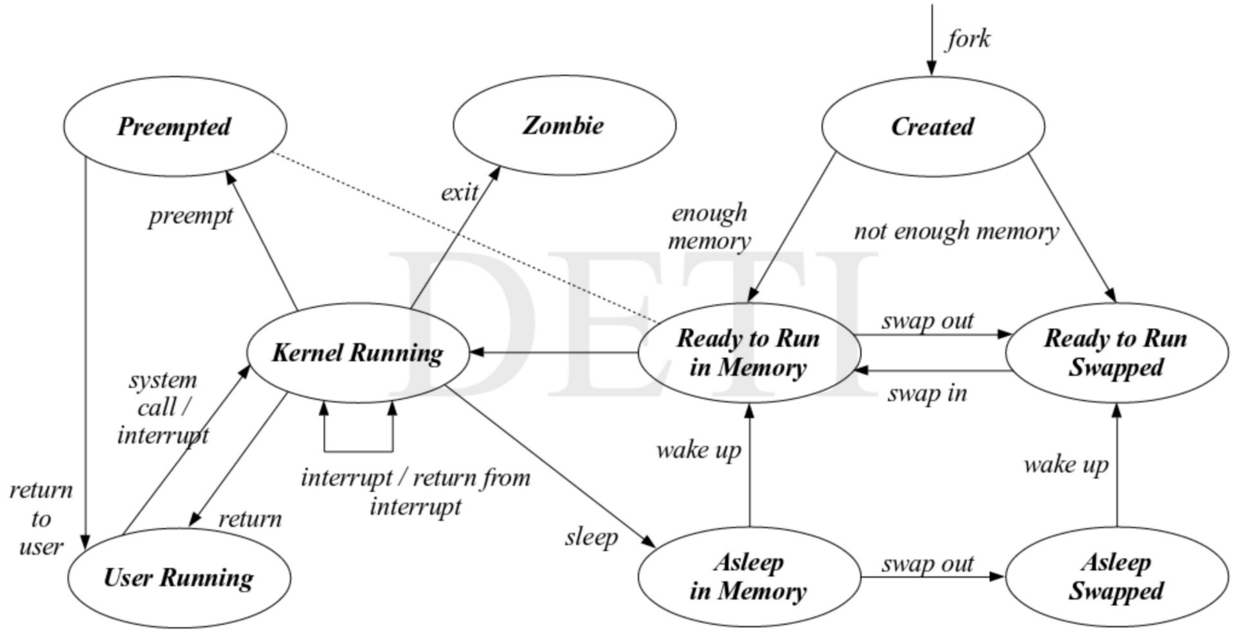
- There are two *run* states, *kernel running* and *user running*, associated to the processor running mode, supervisor and user, respectively
- The *ready* state is also splitted in two states, *ready to run in memory* and *preempted*, but they are equivalent, as indicated by the dashed line

State diagram of a Unix process



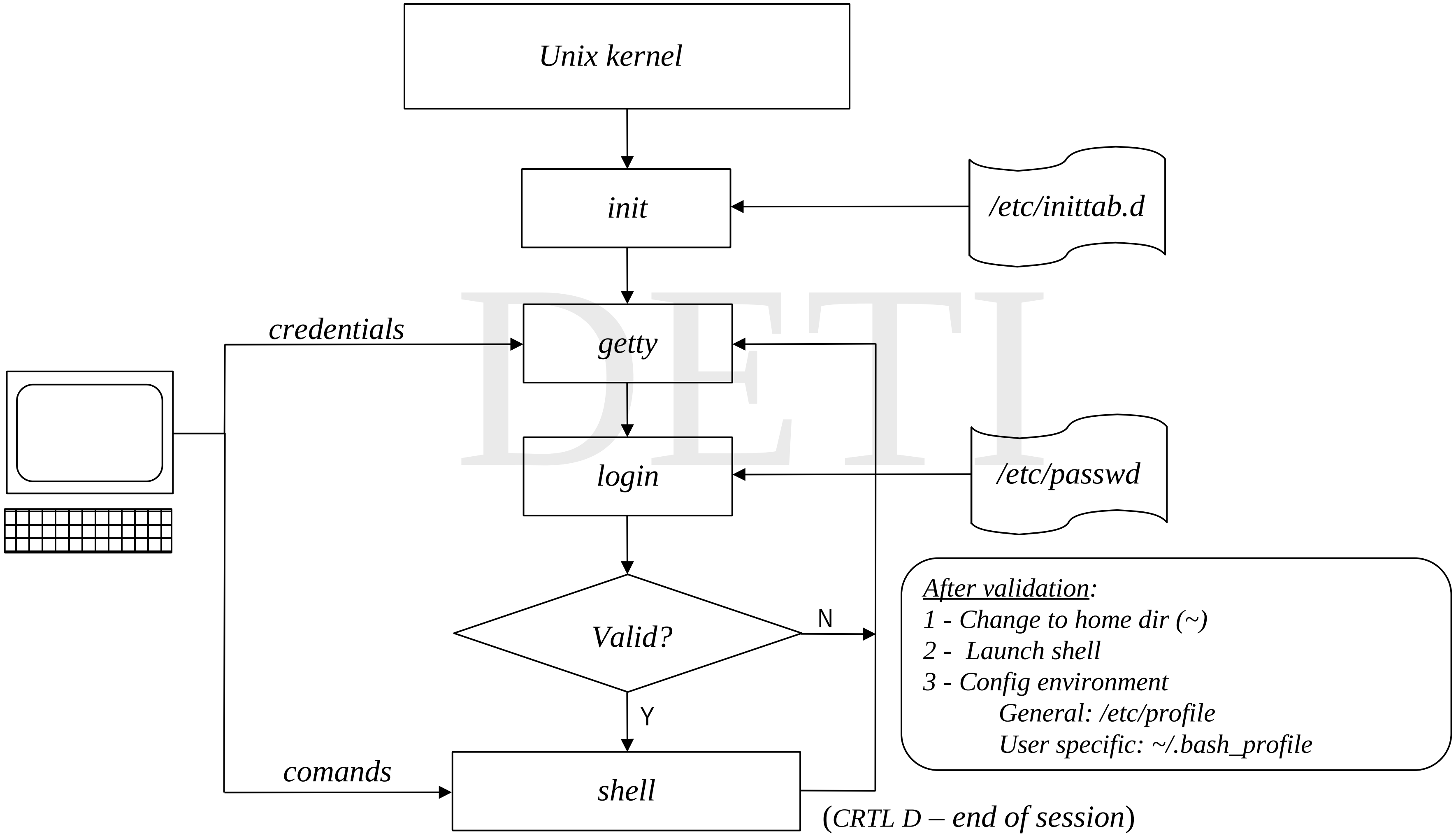
- When a user process leaves supervisor mode, it can be preempted (because a higher priority process is ready to run)
- In practice, processes in *ready to run in memory* and *preempted* shared the same queue, thus are treated as equal
- The *time-out* transition is covered by the *preempt* one

State diagram of a Unix process

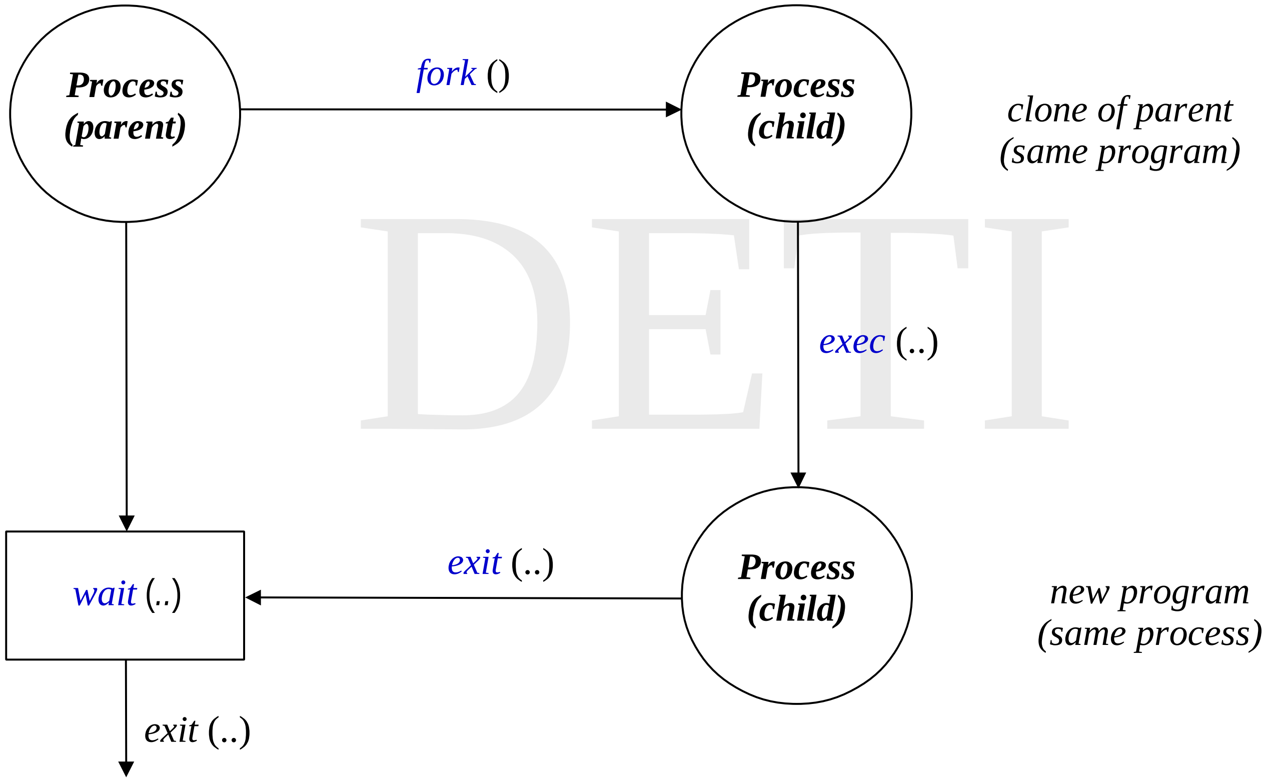


- Tradicionally, execution in supervisor mode could not be interrupted (thus UNIX does not allow real time processing)
- In current versions, namely from SVR4, the problem was solved by dividing the code into a succession of atomic regions between which the internal data structures are in a safe state and therefore allowing execution to be interrupted
- This corresponds to a transition between the *preempted* and *kernel running* states, that could be called *return to kernel*.

Unix – traditional login



Process creation in Unix



Process creation in Unix

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

int main(void)
{
    printf("Before the fork:\n");
    printf("  PID = %d, PPID = %d.\n",
            getpid(), getppid());

    fork();

    printf("After the fork:\n");
    printf("  PID = %d, PPID = %d. Who am I?\n",
            getpid(), getppid());

    return EXIT_SUCCESS;
}
```

fork1.cpp

- The fork clones the executing process, creating a replica of it
- The address spaces of the two processes are equal
  - actually, just after the fork, they are the same
  - typically, a copy on write approach is followed
- The states of execution are the same
  - including the program counter
- Some process variables are different (PID, PPID, ...)

- What can we do with this?

Process creation in Unix

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

int main(void)
{
    printf("Before the fork:\n");
    printf("  PID = %d, PPID = %d.\n",
            getpid(), getppid());

    int ret = fork();

    printf("After the fork:\n");
    printf("  PID = %d, PPID = %d, ret = %d\n",
            getpid(), getppid(), ret);

    return EXIT_SUCCESS;
}
```

fork2.cpp

- The value returned by the fork is different in parent and child processes
  - in parent is the PID of the child
  - in child is always 0
- This return value can be used as a boolean variable, so we can distinguish the code running on child and parent

Process creation in Unix

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

int main(void)
{
    printf("Before the fork:\n");
    printf("  PID = %d, PPID = %d.\n",
            getpid(), getppid());

    int ret = fork();

    if (ret == 0)
    {
        printf("I'm the child:\n");
        printf("  PID = %d, PPID = %d\n",
            getpid(), getppid());
    }
    else
    {
        printf("I'm the parent:\n");
        printf("  PID = %d, PPID = %d\n",
            getpid(), getppid());
    }

    return EXIT_SUCCESS;
}
```

fork3.cpp

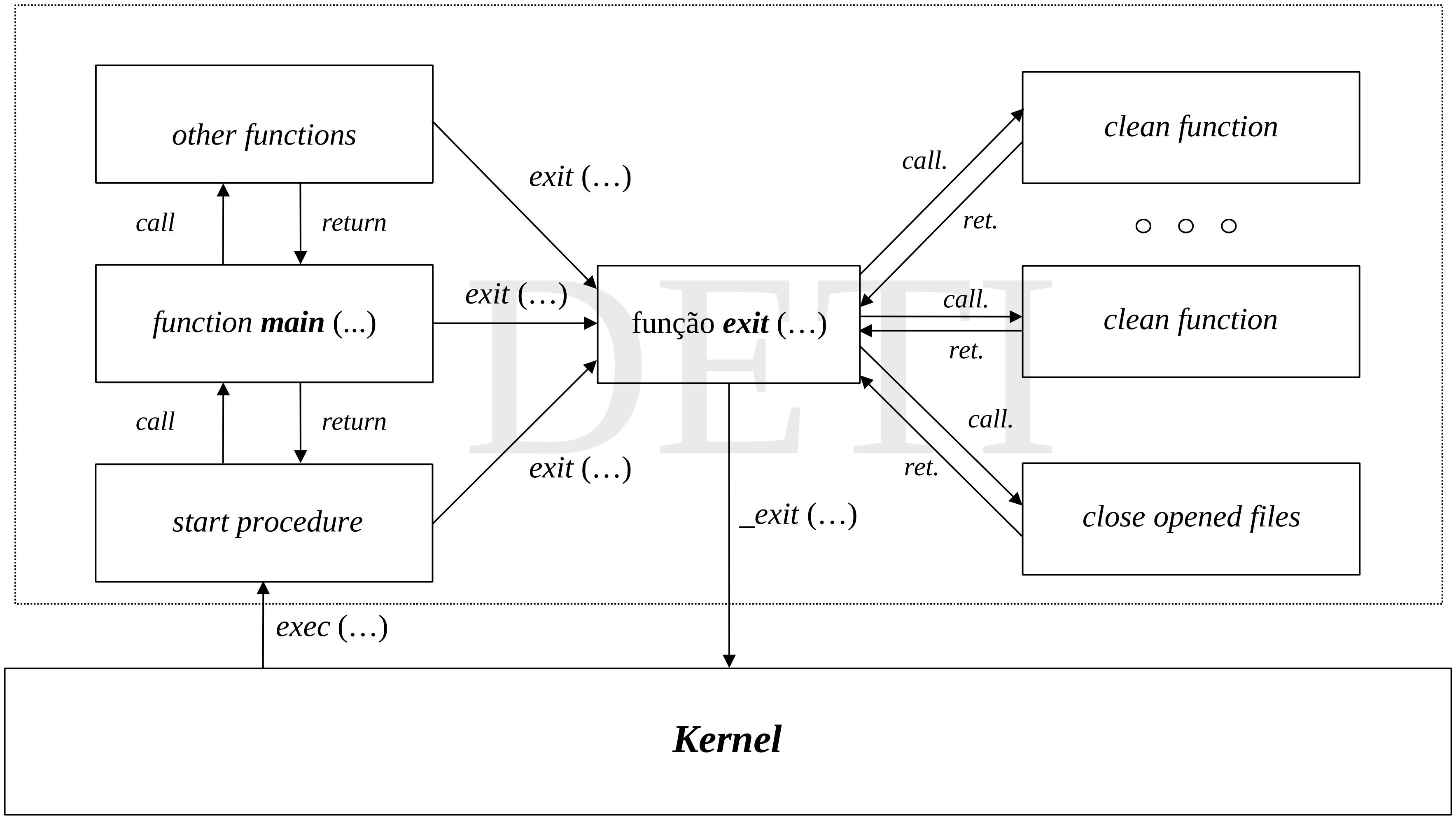
- By itself, the fork is of little interest
- In general, we want to run a different program in the child
  - exec system call
  - there are different versions of exec
- Sometimes, we want the parent to wait for the conclusion of the program running in the child
  - wait system call

Process creation in Unix

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

int main(int argc, char *argv[])
{
    /* check arguments */
    if (argc != 2)
    {
        fprintf(stderr, "spawn <path to file>\n");
        exit(EXIT_FAILURE);
    }
    char *aplic = argv[1];

    printf("===============================\n");

    /* clone phase */
    int pid;
    if ((pid = fork()) < 0)
    {
        perror("Fail cloning process");
        exit(EXIT_FAILURE);
    }

    /* exec and wait phases */
    if (pid != 0) // only runs in parent process
    {
        int status;
        while (wait(&status) == -1);
        printf("=============================\n");
        printf("Process %d (child of %d)"
              " ends with status %d\n",
               pid, getpid(), WEXITSTATUS(status));
    }
    else // this only runs in the child process
    {
        execl(aplic, aplic, NULL);
        perror("Fail launching program");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS); // or return EXIT_SUCCESS
}
```

spawn.cpp

Executing a C/C++ program



Executing a C/C++ program

```
#include   <stdio.h>
#include   <stdlib.h>
#include   <unistd.h>
#include   <assert.h>

/* cleaning functions */
static void atexit_1(void)
{
    printf("atexit 1: %d\n", ++a);
}

static void atexit_2(void)
{
    printf("atexit 2: %d\n", ++a);
}

/* programa principal */
int main(void)
{
    /* registering at exit functions */
    assert(atexit(atexit_1) == 0);
    assert(atexit(atexit_2) == 0);

    /* normal work */
    printf("hello world 1!\n");

    for (int i = 0; i < 5; i++) sleep(1);

    return EXIT_SUCCESS;
}
```

atexit.cpp

- The atexit function allows to register a function to be called at the program's normal termination
- They are called in reverse order relative to their register
- What happens if the termination is forced?

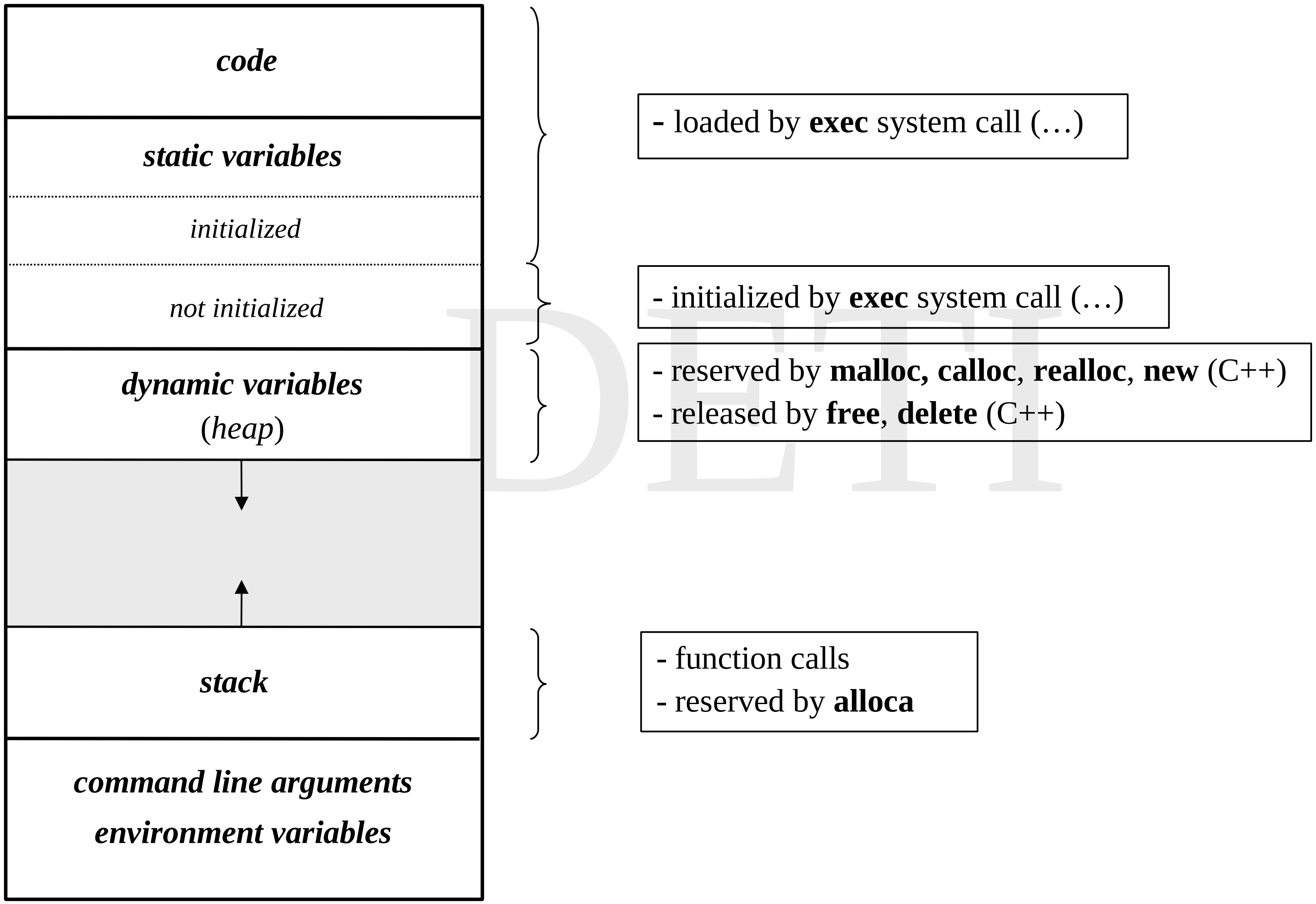
Command line arguments and environment variables

```
#include    <stdio.h>
#include    <stdlib.h>
#include    <unistd.h>

int main(int argc, char *argv[], char *env[])
{
    /* printing command line arguments */
    printf("Command line arguments:\n");
    for (int i = 0; argv[i] != NULL; i++)
    {
        printf("  %s\n", argv[i]);
    }

    /* printing all environment variables */
    printf("\nEnvironment variables:\n");
    for (int i = 0; env[i] != NULL; i++)
    {
        printf("  %s\n", env[i]);
    }

    /* printing a specific environment variable */
    printf("\nEnvironment variable:\n");
    printf("  env[\"HOME\"] = \"%s\"\n", getenv("HOME"));
    printf("  env[\"zzz\"] = \"%s\"\n", getenv("zzz"));

    return EXIT_SUCCESS;
}
```

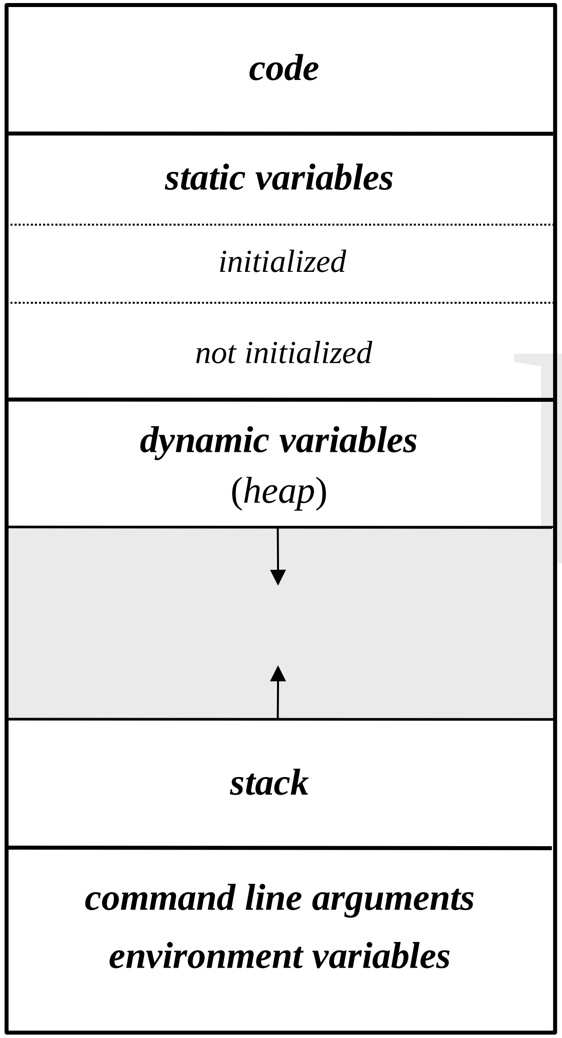
environ.cpp

- `argv` is an array of strings
  - `argv[0]` is the program reference
- `env` is an array of strings, each one representing a variable, in the form name-value pair
- `getenv` returns the value of a variable name

Address space of a Unix process



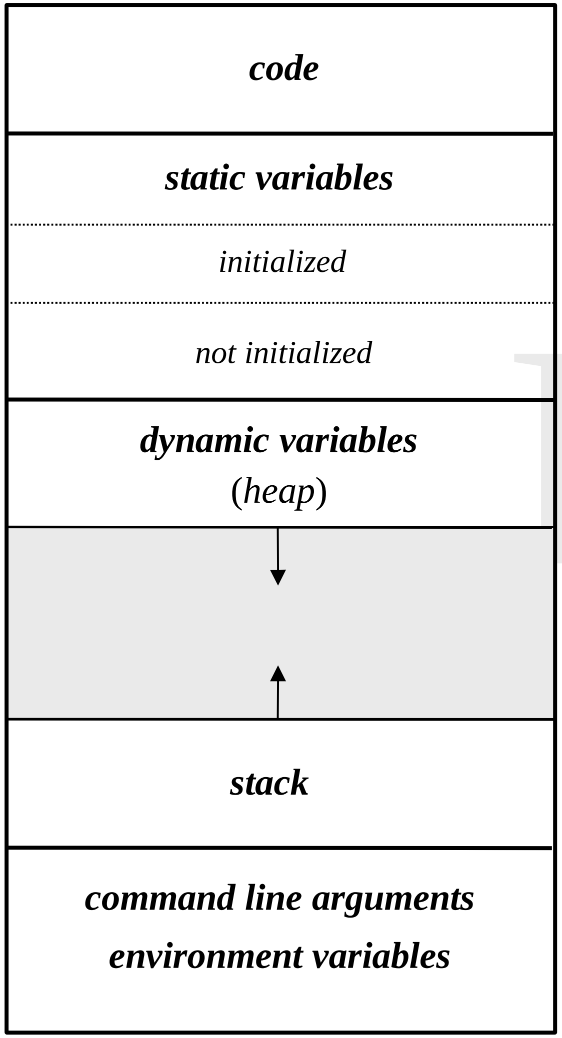
Address space of a Unix process




```
int n1 = 1;
static int n2 = 2;
int n3;
static int n4;
int n5;
static int n6 = 6;

int main(int argc, char *argv[], char *env[])
{
    extern char** environ;
    static int n7 = 3;
    static int n8;
    int *n9 = (int*)malloc(sizeof(int));
    int *n10 = (int*)malloc(sizeof(int));
    int *n11 = (int*)alloca(sizeof(int));
    int n12;
    int n13 = 13;
    int n14;
    printf("\ngetenv(n0) = %p\n", getenv("n0"));
    printf("\nargv = %p\nenviron = %p\nenv = %p\nmain = %p\n\n",
           argv, environ, env, main);
    printf("\n&argc = %p\n&argv = %p\n&env = %p\n",
           &argc, &argv, &env);
    printf("&n1 = %p\n&n2 = %p\n&n3 = %p\n&n4 = %p\n&n5 = %p\n"
           "&n6 = %p\n&n7 = %p\n&n8 = %p\nn9 = %p\nn10 = %p\n"
           "n11 = %p\n&n12 = %p\n&n13 = %p\n&n14 = %p\n",
           &n1, &n2, &n3, &n4, &n5, &n6, &n7, &n8,
           n9, n10, n11, &n12, &n13, &n14);

    return EXIT_SUCCESS;
}
```

mem.cpp

Address space of a Unix process

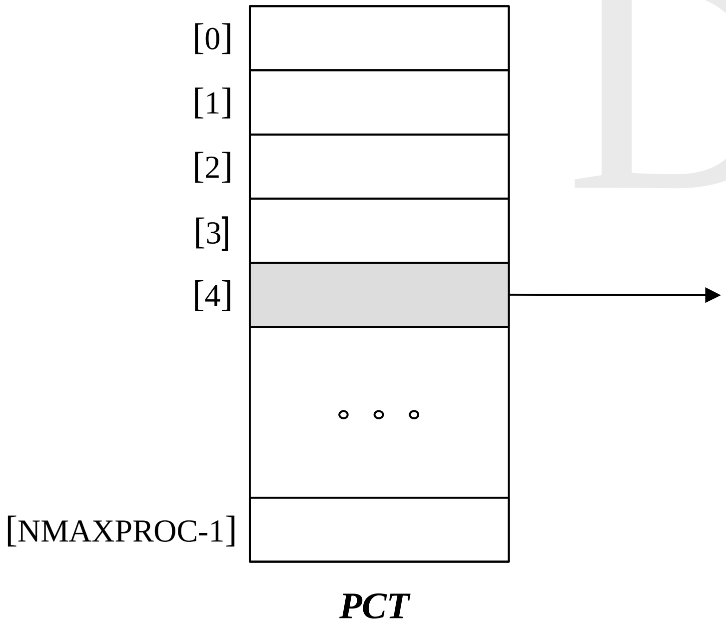



```
int n1 = 1;
...
static int n6 = 6;

int main(int argc, char *argv[], char *env[])
{
    if (fork() != 0)
    {
        fprintf(stderr, "n1 = %d\n", n1);
        wait(NULL);
        fprintf(stderr, "================\n");
        fprintf(stderr, "n1 = %d\n", n1);
        fprintf(stderr, "================\n");
    }
    else
    {
        n1 = 1111;
        fprintf(stderr, "n1 = %d\n", n1);
    }

    static int n7 = 3;
    static int n8;
    ...
    printf("&n1 = %p\n&n2 = %p\n&n3 = %p\n&n4 = %p\n&n5 = %p\n"
           "&n6 = %p\n&n7 = %p\n&n8 = %p\nn9 = %p\nn10 = %p\n"
           "n11 = %p\n&n12 = %p\n&n13 = %p\n&n14 = %p\n",
           &n1, &n2, &n3, &n4, &n5, &n6, &n7, &n8,
           n9, n10, n11, &n12, &n13, &n14);

    return EXIT_SUCCESS;
}
```

mem-fork.cpp

Process control table

- To implement the process model, the operating systems needs a data structure to be used to store the information about each process – process control block
- The process control table (PCT), an array of process control blocks, stores information about all processes



identification – of process, parent, user and user's group

address space – location of code, data and stack (in memoty or swapping area)

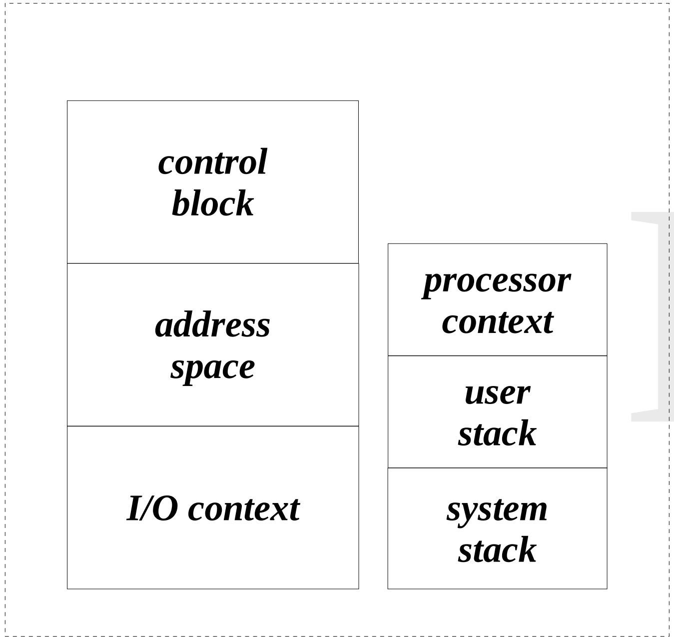
processor context – internal registers (including program counter and stack pointer) when the switching occurs

I/0 context – communication channels and associated buffers

state - state of execution (run, ready, blocked, ...), priority, event

statistic - start time, CPU time, ...

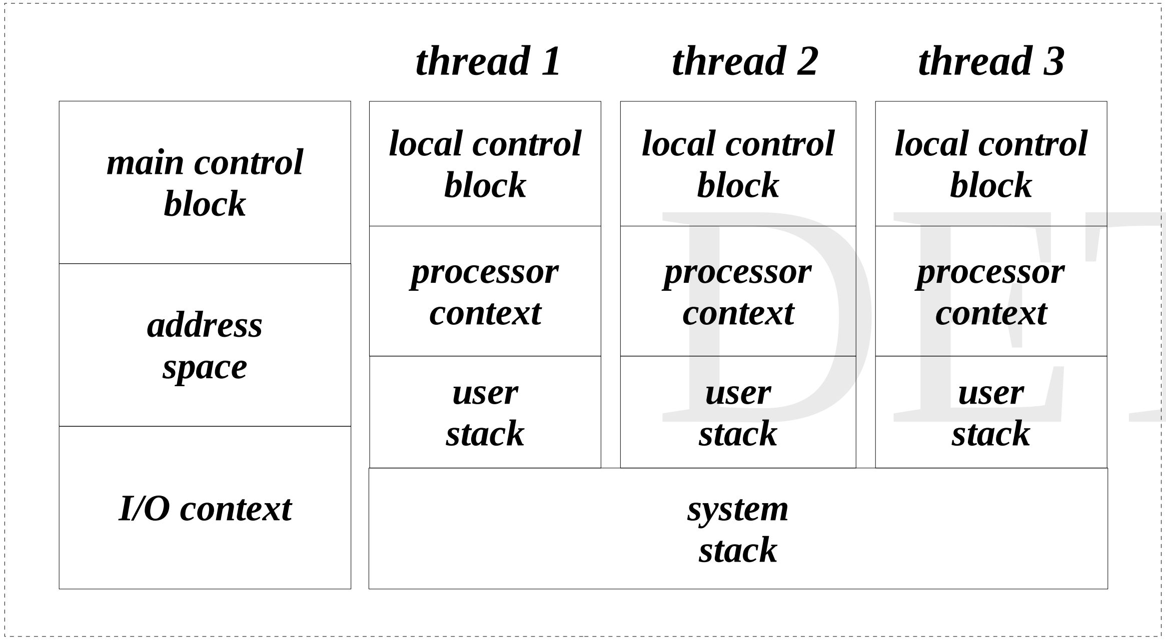
Threads



Single threading

- In traditional operating system, a process includes:
  - an address space and a set of communication channels with I/O devices
  - a single thread of control, which incorporates the processor registers (including the program counter) and a stack
- However, these components can be managed separetely
- In this model, ***thread*** appears as an execution component within a process

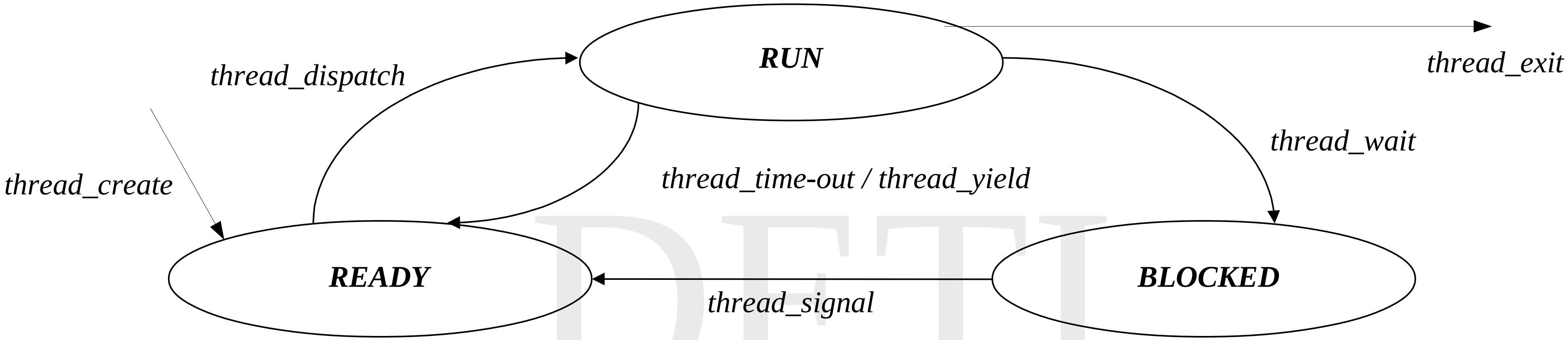
Multithreading



Multithreading

- Several independent threads can coexist in the same process, thus sharing the same address space and the same I/O context
  - This is referred to as *multithreading*
- In practice, threads can be seen as *light weight processes*

State diagram of a thread

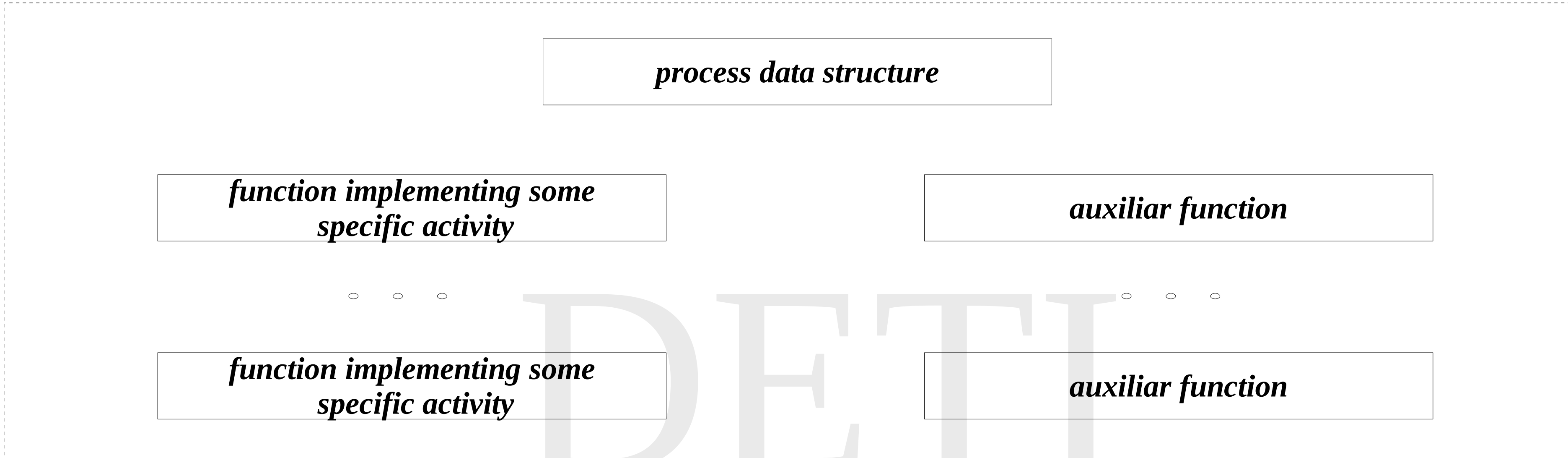


- only states concerning the management of the processor are considered
- states SUSPENDED-READY and SUSPENDED-BLOCKED are not present, since they are related to the address space and thus are related to the process, not to the threads
- states NEW and TERMINATED are not present, since the management of the multiprogramming environment is basically related to restrict the number of threads that can exist within a process

Advantages of multithreading

- *easier implementation of applications* – in many applications, decomposing the solution into a number of parallel activities makes the programming model simpler
  - since the address space and the I/O context is shared among all threads, multithreading favors this decomposition.
- *better management of computer resources* – creating, destroying and switching threads is easier then doing the same with processes
- *better performance* – when an application envolves substantial I/O, multithreading allows activities to overlap, thus speeding up its execution
- *multiprocessing* – real parallelism is possible if multiples CPUs exist.

Structure of a multithreaded program

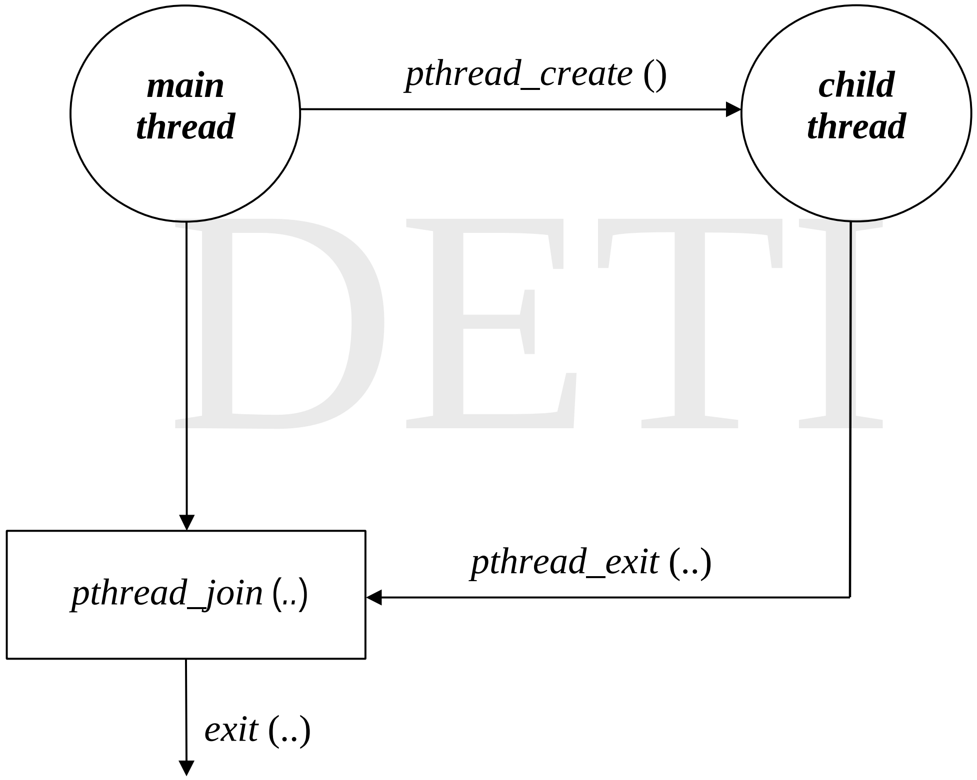


- each thread is typically associated to the execution of a function that implements some specific activity
- communication between threads can be done through the process data structure, which is global from the threads point of view
- the main program, also represented by a function that implements a specific activity, is the first thread to be created and, in general, the last to be destroyed

Implementations of multithreading

- *user level threads* – *threads* are implemented by a library, at user level, which provides creation and management of threads without kernel intervention
  - versatile and portable
  - when a thread calls a blocking system call, the whole process blocks
    - because the kernel only sees the proccess
- *kernel level threads* – *threads* are implemented directly at kernel level
  - less versatile and portable
  - when a thread calls a blocking system call, another thread can be schedule to execution

Library pthread



Library pthread

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

/* return status */
int status;

/* child thread */
void *threadChild (void *par)
{
  printf ("I'm the child thread!\n");

  status = EXIT_SUCCESS;
  pthread_exit (&status);
}
```

```
/*  main thread */
int main (int argc, char *argv[])
{
  /* launching the child thread */
  pthread_t thr;
  if (pthread_create (&thr, NULL,
                threadChild, NULL) != 0)
  {
    perror ("Fail launching thread");
    return EXIT_FAILURE;
  }
  /* waits for child termination */
  if (pthread_join (thr, NULL) != 0)
  {
    perror ("Fail joining child");
    return EXIT_FAILURE;
  }

  printf ("Child ends; status %d.\n", status);

  return EXIT_SUCCESS;
}
```

thread.cpp

Threads in Linux

- In Linux there are two system calls to create a child process:
  - *fork* – creates a new process that is a full copy of the current one
    - the address space and I/O context are duplicated
  - *clone* – creates a new process that can share elements with its parent
    - address space, table of file descriptors, and table of signal handlers, for example, are shareable.
    - the child starts execution in a given function
- Thus, from the kernel point of view, processes and threads are treated similarly
- Threads of the same process forms a thread group and have the same thread group identifier (TGID)
  - this is the value returned by system call `getpid`()
- Within a group, threads can be distinguished by their unique thread identifier (TID)
  - this value is returned by system call `gettid`()

Threads in Linux

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

pid_t gettid()
{
    return syscall(SYS_gettid);
}

/* child thread */
int status;
void *threadChild (void *par)
{
  printf ("Child: PPID: %d, PID: %d, TID: %d\n",
            getppid(), getpid(), gettid());

  status = EXIT_SUCCESS;
  pthread_exit (&status);
}

...
```

tid.cpp

- there is not glibc wrapper for `gettid`
  - it has to be called indirectly, using system call `syscall`
- The TID of the main thread is the same as the PID of the process
  - actually, they are the same entity

Process switching

- Current processors have two functioning modes:
  - *supervisor mode* – all instruction set can be executed
    - is a privileged mode, reserved to the operating system
  - *user mode* – only part of the instruction set can be executed
    - input/output instructions are excluded as well as those that modify control registers
    - it is the normal mode of operation
- Switching from user mode to supervisor mode is only possible through an *exception* (for security reasons)
- An exception can be caused by:
  - I/O interrupt
  - illegal instruction (division by zero, bus error)
  - trap instruction (software interruption)

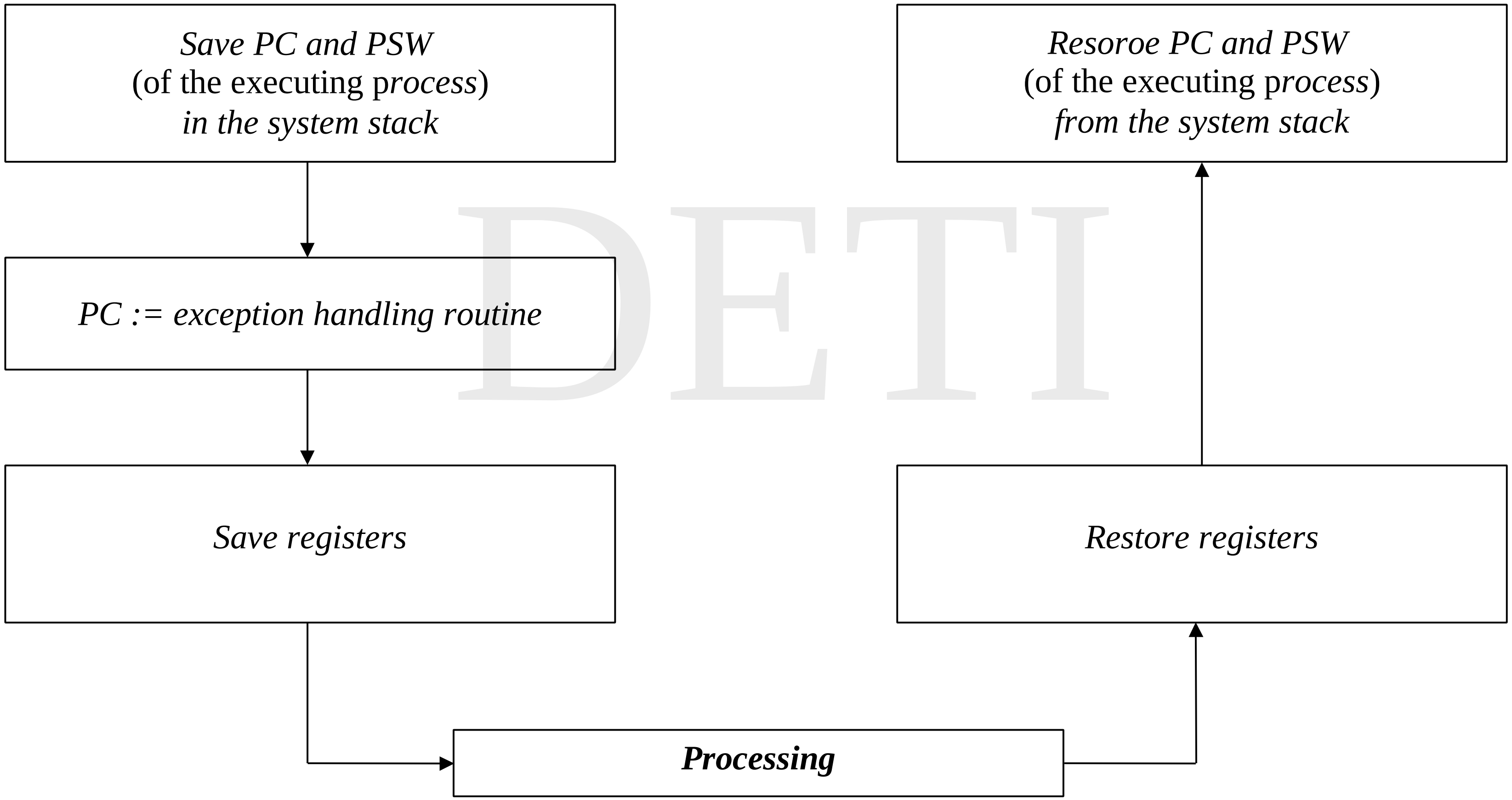
Process switching

- The operating system should function in supervisor mode
  - in order to have access to all the functionalities of the processor
- Thus kernel functions (including system calls) must be fired by
  - hardware (interrupt)
  - trap (software interruption)
- There is a uniform operation environment: *exception handling*

- Process switching can be seen the same way, with a small difference

Process switching

Processing a (normal) exception



Process switching

Processing a process switching

Save data of running process
(address space, processor registers, I/O data, ...)
in PCT

Update process state
(also in PCT)

time-out → READY
wait event → BLOCKED
exit / abort → TERMINATED

Select a process in the READY state
(according to some police)

Update state of selected process
(in PCT)

Restore date of selected process
(address space, processor state, I/O data, ...)
from PCT

Bibliography

Operating Systems Concepts; Silberschatz, Galvin & Gagne; John Wiley & Sons, 9th Ed

- Chapter 3: *Processes* (sections 3.1 to 3.3)
- Chapter 4: *Threads* (sections 4.1 and 4.4.1)

Modern Operating Systems; Tanenbaum & Bos, Prentice-Hall International Editions, 4rd Ed

- Chapter 2: *Processes and Threads* (sections 2.1 and 2.2)

Operating Systems, W. Stallings, Prentice-Hall International Editions, 7th Ed

- Chapter 3: *Process Description and Control* (sections 3.1 to 3.5 and 3.7)
- Chapter 4: *Threads* (sections 4.1, 4.2 and 4.6)