



Sistemas de Operação / Fundamentos de Sistemas Operativos

Processes in Unix/Linux

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

Outline

- ① Program vs. Process
- ② Process in Unix/Linux

Process

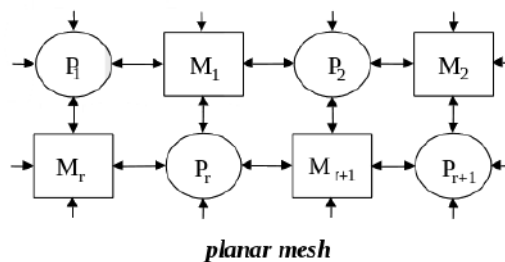
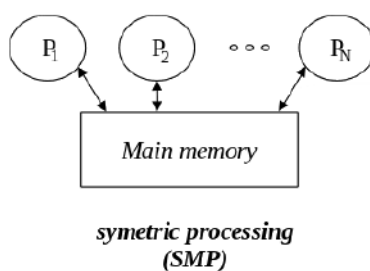
Program vs. process

- **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:
 - **addressing space** – code and data (actual values of the different variables) of the associated program
 - input and output data (data that are being transferred from input devices and to output devices)
 - process specific variables (PID, PPID, ...)
 - actual values of the processor internal registers
 - state of execution
- Different processes can be running the same program
- In general, there are more processes than processors – **multiprogramming**

Multiprocessing vs. Multiprogramming

Multiprocessing

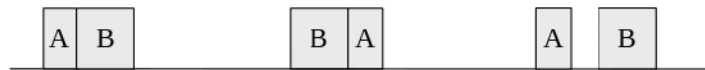
- **Parallelism** – ability of a computational system to simultaneously run two or more programs
 - more than one processor is required (one for each simultaneous execution)
- The operating systems of such computational systems supports **multiprocessing**



Multiprocessing vs. Multiprogramming

Multiprogramming

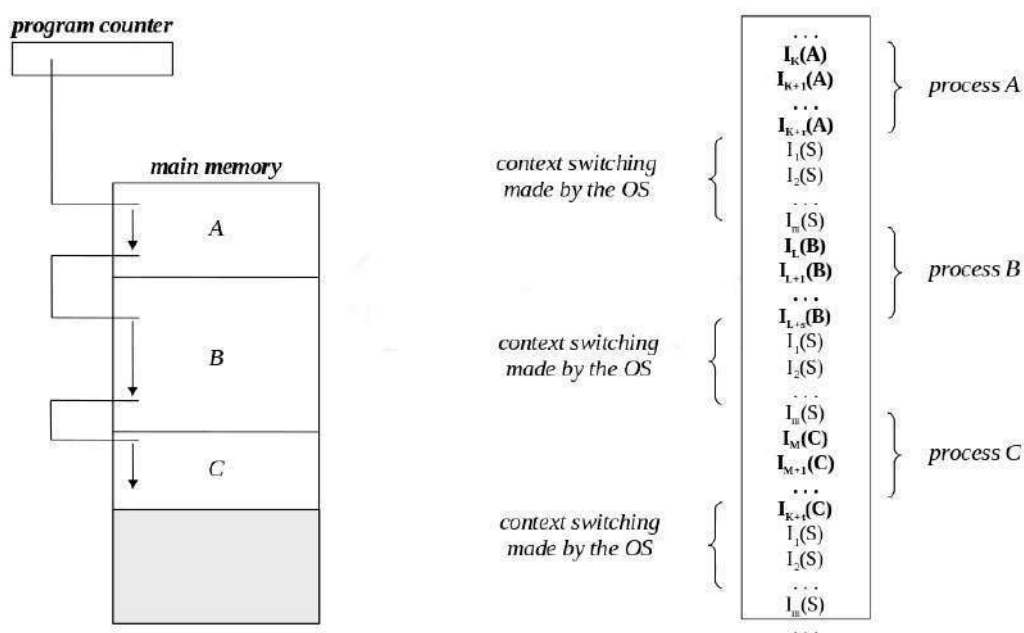
- **Concurrency** – illusion created by a computational system of apparently being able to simultaneously run more programs than the number of existing processors
- The existing processor(s) must be assigned to the different programs in a time multiplexed way
- The operating systems of such computational systems supports **multiprogramming**



- Programs A and B are executing concurrently in a single processor computational system

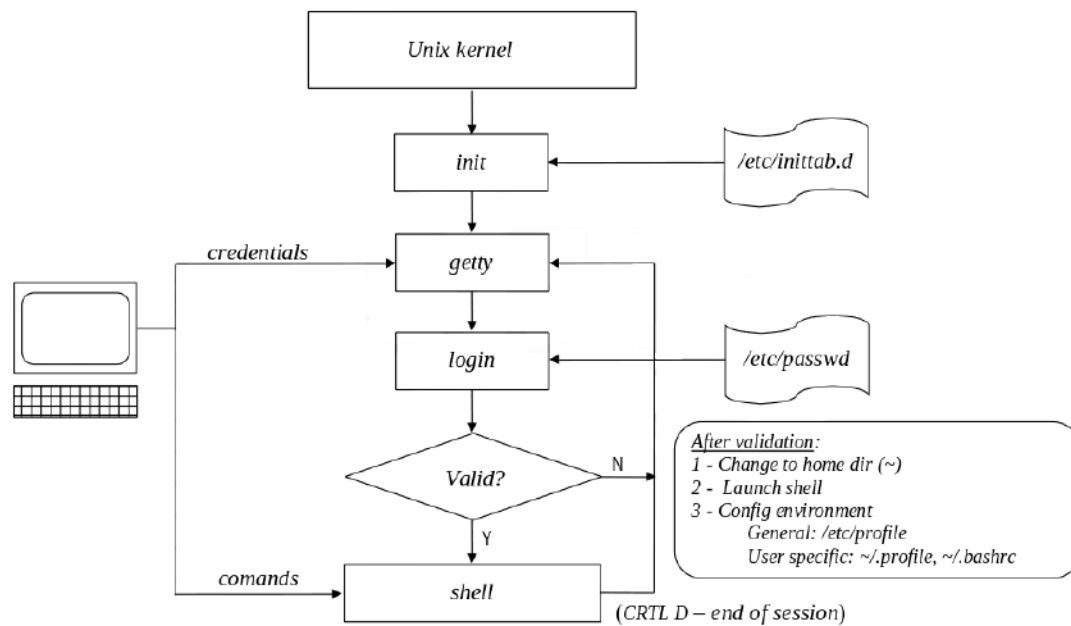
Process

Execution in a multiprogrammed environment



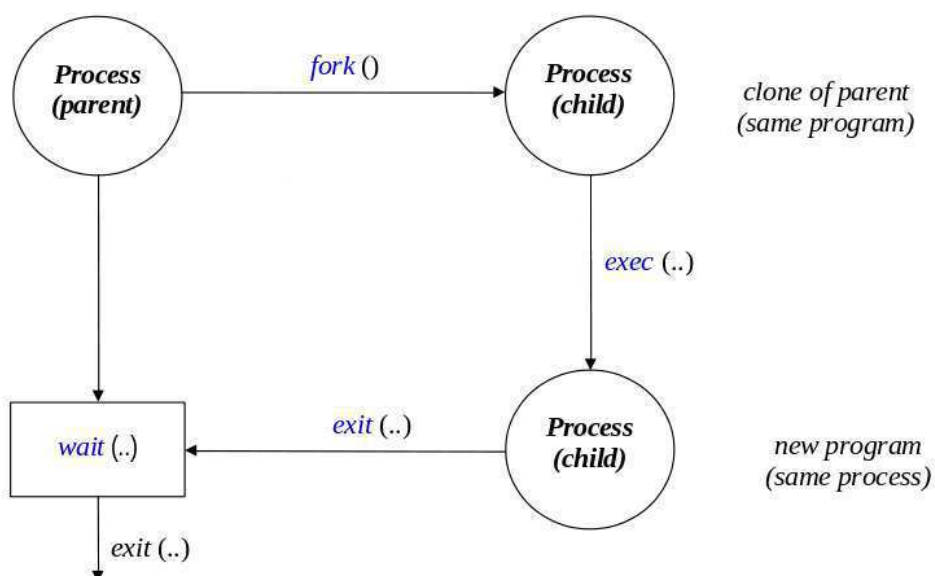
Processes in Unix

Traditional login



Processes in Unix

Creation by cloning



Processes in Unix

Process creation: `fork1`

```
#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.\n",
        "  Am I the parent or the child?"
        "  How can I know it?\n",
        getpid(), getppid());

    return EXIT_SUCCESS;
}
```

- 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?

Processes in Unix

Process creation: `fork2` and `fork3`

```
#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.\n",
        getpid(), getppid());
    printf("  ret = %d\n", ret);

    return EXIT_SUCCESS;
}
```

- The value returned by the fork is different in parent and child processes
 - in the parent, it is the PID of the child
 - in the child, it is always 0

Processes in Unix

Process creation: `fork2` and `fork3`

```
#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;
}
```

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

Processes in Unix

Process creation: `fork3`

```
#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;
}
```

- In general, used alone, 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
- *In this code, we are assuming the fork doesn't fail*
 - in case of an error, it returns -1

Process creation in Unix

Launching a program: `fork` + `exec`

```
#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, "launch <<cmd>>\n");
        exit(EXIT_FAILURE);
    }
    char *aplic = argv[1];

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

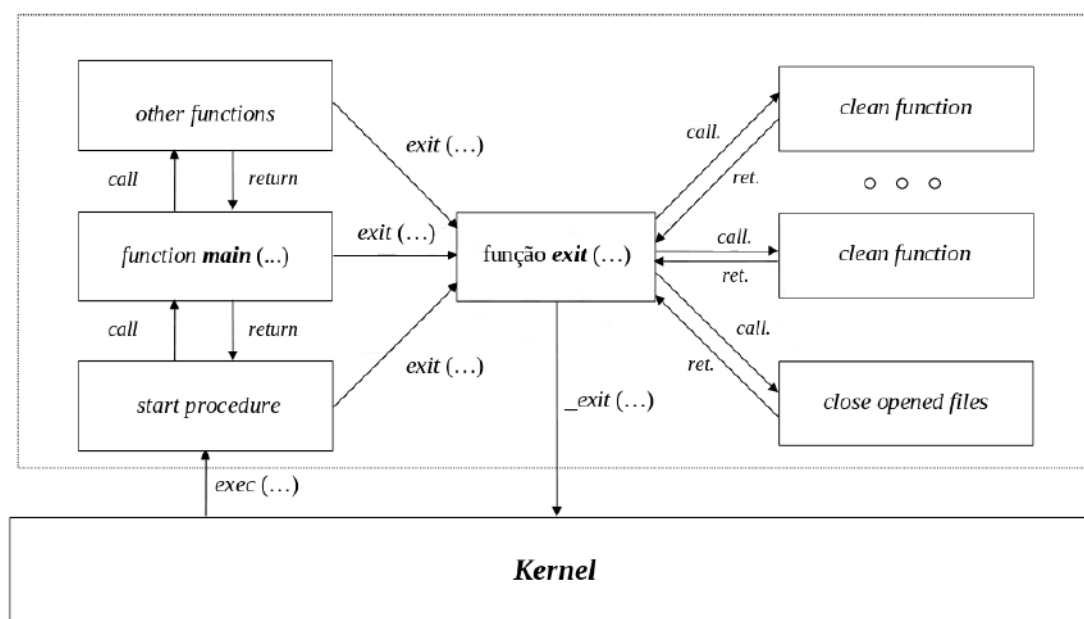
    /* clone phase */
    int pid;
    if ((pid = fork()) < 0)
    {
        perror("Fail cloning process");
        exit(EXIT_FAILURE);
    }

    /* 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
}
```

Processes in Unix

Execution of a C/C++ program



Processes in Unix

Executing a C/C++ program: `atexit`

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

/* cleaning functions */
static void atexit_1(void)
{
    printf("atexit 1\n");
}

static void atexit_2(void)
{
    printf("atexit 2\n");
}

/* main programa */
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;
}
```

- 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?*

Processes in Unix

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]);
    }

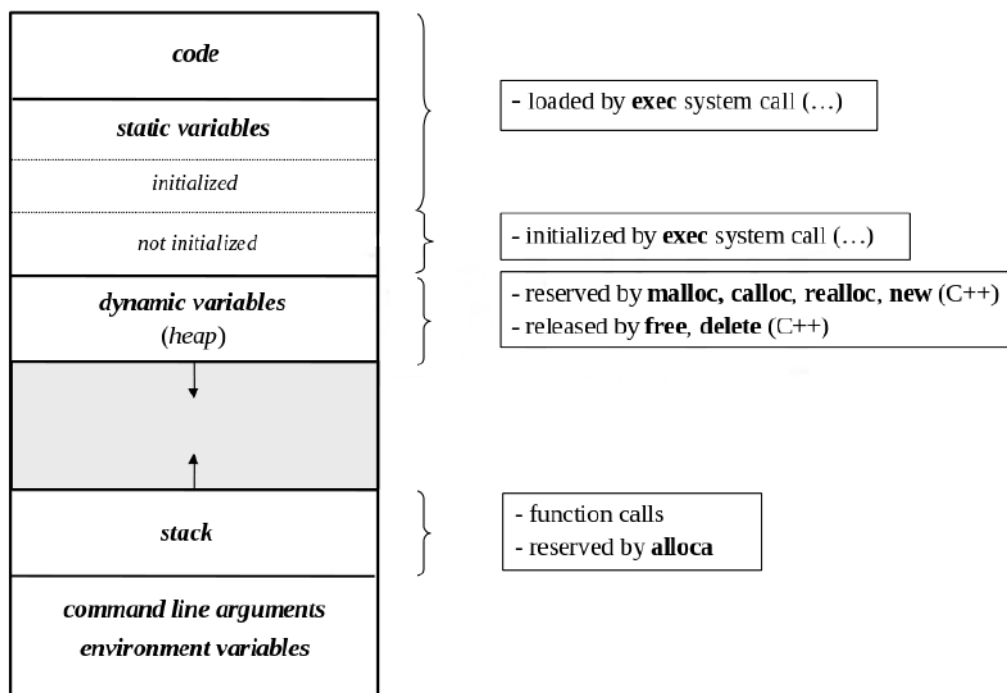
    /* printing a specific environment variable */
    printf("\nEnvironment variable:\n");
    printf(" env[\"HOME\"] = \"%s\"\n", getenv("HOME"));
    printf(" env[\"zzz\"] = \"%s\"\n", getenv("zzz"));

    return EXIT_SUCCESS;
}
```

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

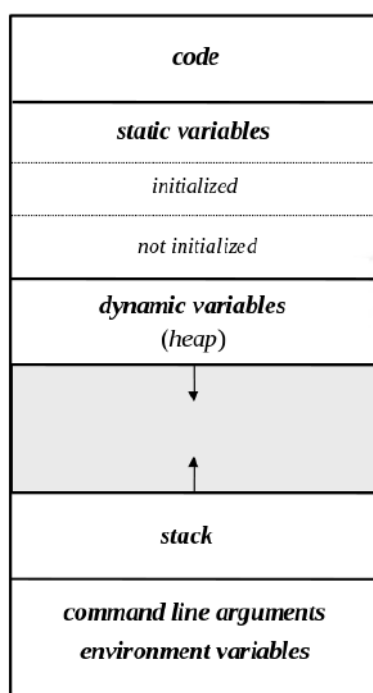
Processes in Unix

Address space of a Unix process



Processes in Unix

Address space of a Unix process (2)

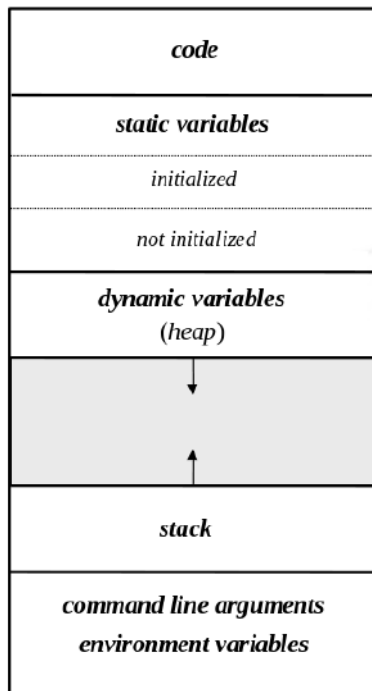


```
int n1 = 1;
static int n2 = 2;
int n3;
static int n4;
int n5;
static int n6 = 6;

int main(int argc, char *argv[], char *env[])
{
    extern char** environ;
    static int n7;
    static int n8 = 8;
    int *p9 = (int*) malloc(sizeof(int));
    int *p10 = new int;
    int *p11 = (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",
        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\n&n9: %p\n&n10: %p\n"
        "&p11: %p\n&n12: %p\n&n13: %p\n&n14: %p\n",
        &n1, &n2, &n3, &n4, &n5, &n6, &n7, &n8,
        p9, p10, p11, &n12, &n13, &n14);
}
```

Processes in Unix

Address space of a Unix process (3)



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

int n01 = 1;

int main(int argc, char *argv[], char *env[])
{
    int pid = fork();
    if (pid != 0)
    {
        fprintf(stderr, "%5d: n01 = %5d (%p)\n",
            pid, n01, &n01);
        wait(NULL);
        fprintf(stderr, "%5d: n01 = %5d (%p)\n",
            pid, n01, &n01);
    }
    else
    {
        fprintf(stderr, "%5d: n01 = %5d (%p)\n",
            pid, n01, &n01);
        n01 = 1111;
        fprintf(stderr, "%5d: n01 = %5d (%p)\n",
            pid, n01, &n01);
    }
    return 0;
}
```



Sistemas de Operação / Fundamentos de Sistemas Operativos

Introduction to operating systems

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

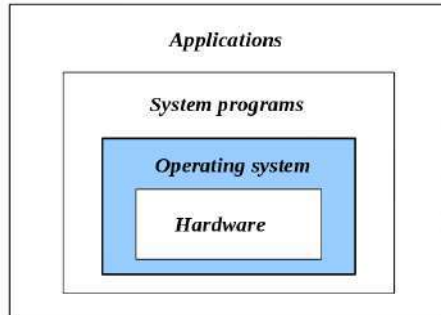
Outline

- 1 Overview
- 2 Evolution of computational systems and operating systems
- 3 Key topics of an operating system

Global view

Purpose of an Operating System

- Support (base) program executed by the computational system



- Gives **life** to the computer (hardware), providing an abstract interaction environment
- Acts as an interface between the machine and the application programs
- Dynamically allocates shared system resources to the executing programs
- Manages and schedules memory, processors, and other system devices

- Objectives of an Operating System:

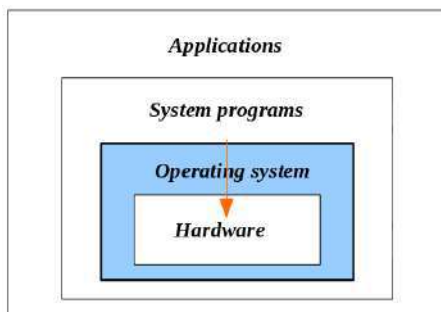
- **Convenience** in the way the computer is used
- **Efficiency** in the use of computer resources
- **Ability to evolve** as to permit the development of new system functions

- Two different perspectives: top-down and bottom-up

Global view

Operating system as an extended machine

- This is a outer-inner or user view



- The **operating system** provides an **abstract view of the underlying computational system** that frees programmers of the hardware details
- A **functional model** of the computational system (a virtual machine), that is simple to understand and program
- The interface with the hardware is a uniform programming environment, defined as a set of **system calls**, that allows the portability of applications among structurally different computational systems

Global view

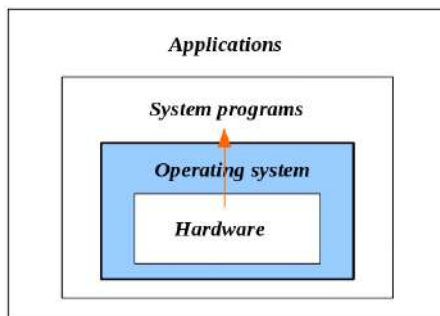
Operating system as an extended machine (2)

- Some typical functionalities of the operating system:
 - Establishment of a user interaction environment
 - Provision of facilities for the development, testing and validation of programs
 - Provision of mechanisms for the controlled execution of programs, including their intercommunication and synchronization
 - Dissociation of the program's address space from the constraints imposed by the size of the main memory
 - Control access to the system as a whole and to specific system resources, protecting against unauthorized access and resolving conflicts
 - Organization of secondary memory in file systems and control access to files
 - Definition of a general model for access to input/output devices, regardless of their specific characteristics
 - Detection of error situations and establishment of appropriate response

Global view

Operating system as a resource manager

- This is the inner-outer or computational system view
- A **computational system** is a system composed of a set of resources (processor(s), main memory, secondary memory, I/O device controllers) targeting the processing and storage of information



- The operating system is the program that manages the computer system, making the controlled and orderly allocation of its different resources to the programs that compete for them
 - resource usage is multiplexed in space and time
- aims at maximizing the performance of the computational system, ensuring the most efficient use of existing resources

Global view

Operating system as a resource manager (2)

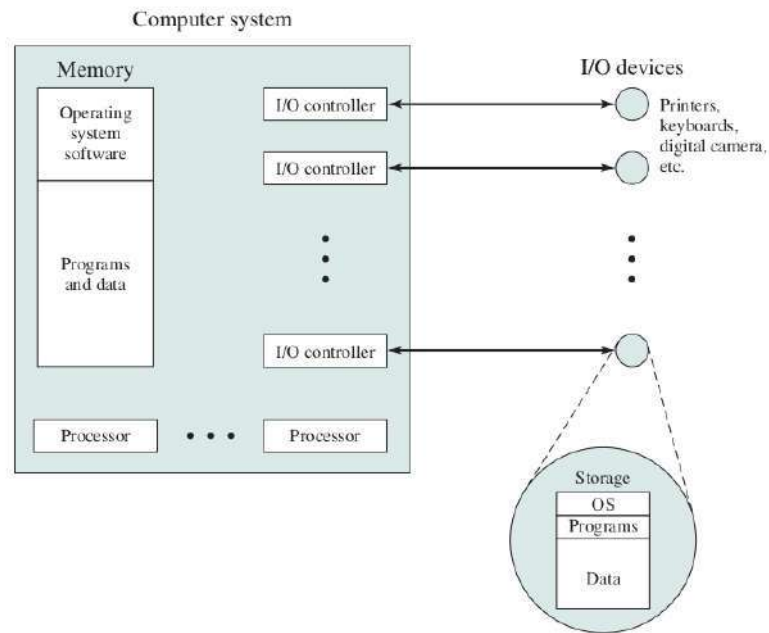


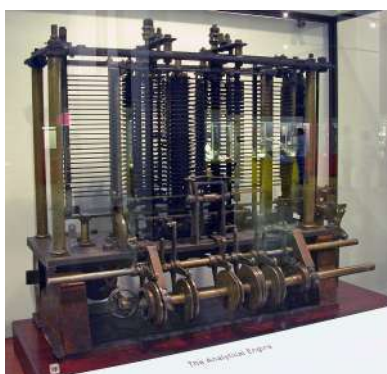
Figure 2.2 The Operating System as Resource Manager

Operating Systems: Internals and Design Principles, William Stallings

Evolution of computational systems

Prehistory

Period	Technology	Features
<ul style="list-style-type: none"> Mid 19th century 	<ul style="list-style-type: none"> mechanical device 	<ul style="list-style-type: none"> no operating system programmable via punched cards conditional statement never work properly

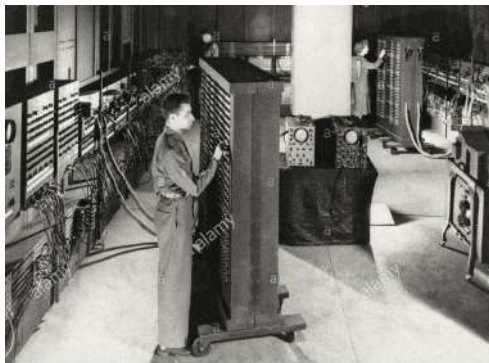


Analytical engine
Charles Babbage (& Ada Lovelace)

Evolution of computational systems

1st generation

Period	Technology	Features
<ul style="list-style-type: none">• 1945–1955 <p>Predecessors</p> <ul style="list-style-type: none">• Atanasoft-Berry (1937)• Konrad Zuse (1941)• Howard Aiken (1944)	<ul style="list-style-type: none">• vacuum tubes• electromechanical relays	<ul style="list-style-type: none">• programmed in machine language• punched cards• serial processing• programmer has full control of the machine• no operating system



ENIAC
(Electronic Numerical Integrator and Calculator)
J. Presper Eckert & John Mauchly
University of Pennsylvania

An Illustrated History of Computers, John Kopplin, 2002

Evolution of operating systems

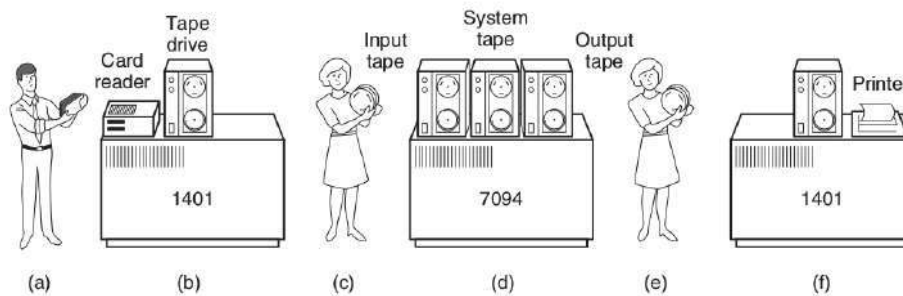
Serial processing

- **Serial processing** – one user at a time
- User has direct access to the computer/processor
 - actually there is no operating system
- Scheduling and setup time issues:
 - computer must be previously reserved by users, possible causing waste of time or premature exit
 - a considerable amount of time was spent setting up the program to run
- Common software for all users (like linkers, libraries, ...) appear
- Very poor processor utilization

Evolution of computational systems

2nd generation

Period	Technology	Features
<ul style="list-style-type: none">• 1955–1965	<ul style="list-style-type: none">• transistor devices	<ul style="list-style-type: none">• FORTRAN and assembly• simple batch operating system (monitor)• rudimentary command language (job control language)

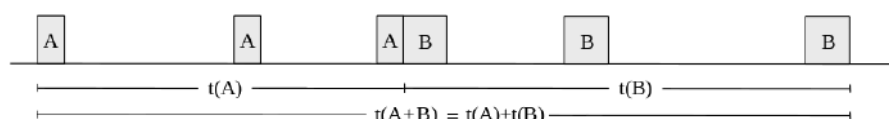


Modern Operating Systems, Andrew Tanenbaum & Herbert Bos

Evolution of operating systems

Simple batch

- Simple batch – one job at a time
- user loses direct access to the processor
 - users submit jobs on cards or tapes to an operator
 - the operator joins jobs into a batch and places it in the computer input device
 - the monitor (batch OS) manages the execution of the batch's jobs in the processor
- better than serial processing
 - scheduling is done by the monitor, one job after the other
 - a job control language helps improving setup time
- requirements like memory protection and privileged instructions appear
 - (most of) the monitor and the user program must be in memory at the same time
- still poor processor utilization
 - processor is often idle



Evolution of computational systems

3rd generation

Period	Technology	Features
• 1965–1980	<ul style="list-style-type: none">• integrated circuits (SSI/MSI)• family of computers (IBM S/360)• 16-bit minicomputers (DEC PDP-n)• 4-, 8-, 12- and 16-bit microprocessors	<ul style="list-style-type: none">• multiprogrammed batch operating system• interactive systems (time-sharing): CTSS (MIT); MULTICS; Unix• (proprietary) general usage operating systems• real-time systems



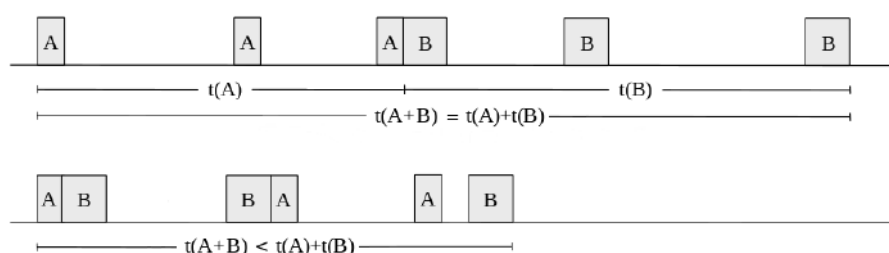
An IBM System/360 Model 20 CPU with front panels removed, with IBM 2560 MFCM (Multi-Function Card Machine)

wikipedia.com

Evolution of operating systems

Multiprogrammed batch

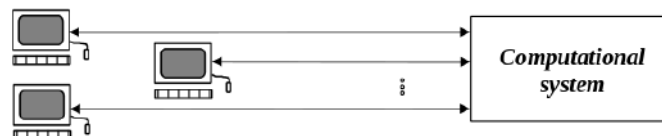
- **Multiprogrammed batch** – while a job is waiting for the conclusion of an input/output operation, another job can use the processor
 - the approach is known as **multiprogramming** or **multitasking**
- scheduling, resource management and hardware support are required
 - monitor and all user programs (jobs) must be simultaneously in memory
 - interrupt and DMA I/O are needed to accomplish the I/O operation
 - what job to dispatch next?
- good processor utilization
 - but there is no direct user interaction with the computer



Evolution of operating systems

Time-sharing

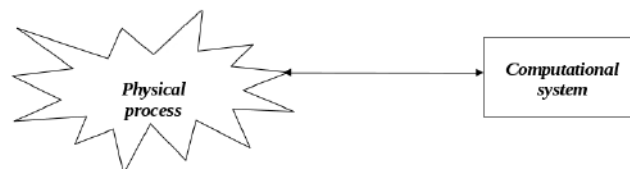
- **Time-sharing** – keeping different users, each one in a different terminal, in direct and simultaneous connection with the system
 - when computers were big and costly, the option was to share them
- Using multiprogramming, the processor is successively assigned to the execution of the various programs present during very short time intervals (time quantum)
 - since a human response, when compared to a computer, is slow, the illusion that the system is entirely dedicated to every user is created
- Differences with multiprogrammed batch
 - try to **minimize response time** instead of **maximize processor use**
 - directives to OS given by **user commands** instead of **job control language commands**
- In addition to memory protection and privileged instructions, file access protection and resource (printers, ...) contention are issues to be considered



Evolution of operating systems

Real-time operating systems

- **Purpose** – use the computer in the online monitoring and control of physical processes
- **Method** – variant of an interactive system that allows to impose maximum limits to the response times of the computational system to different classes of external requests



Evolution of computational systems

4th and 5th generations

4th generation

Period	Technology	Features
<ul style="list-style-type: none">• 1980–present	<ul style="list-style-type: none">• LSI/VLSI• personal computers (microcomputers)• network	<ul style="list-style-type: none">• standard operating systems (MS-DOS, Macintosh, Windows, Unix)• network operating systems

5th generation

Period	Technology	Features
<ul style="list-style-type: none">• 1990–present	<ul style="list-style-type: none">• broadband, wireless• system on chip• smartphone	<ul style="list-style-type: none">• mobile operating systems (Symbian, iOS, Android)• cloud computing• ubiquitous computing

Evolution of operating systems

Network operating systems

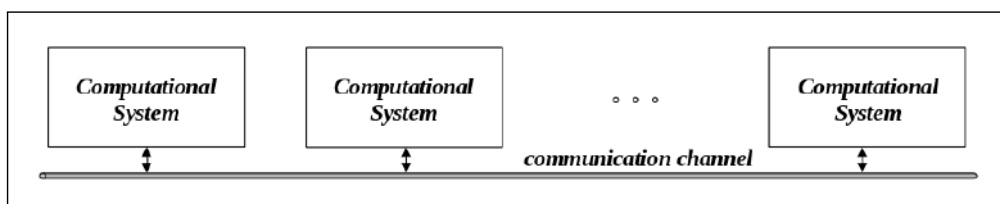
- **Purpose** – take advantage of the interconnection facilities of computer systems (at the hardware level) to establish a set of services common to an entire community
- **Services** – file transfer (ftp); access to remote file system (NFS); resource sharing (printers, etc.); access to remote computational systems (telnet, remote login, ssh); e-mail; access to the world wide web (browsers)



Evolution of operating systems

Distributed operating systems

- **Purpose** – take advantage of the facilities for constructing multiprocessor computing systems, or for interconnecting different computational systems, to establish an integrated interaction environment where the user views the parallel computing system as a single entity
- **Methodology** – ensure a complete transparency in the access to processors or other resources of the distributed system, in order to allow
 - static/dynamic load balancing
 - increasing the speed of processing by incorporation of new processors or new computational systems
 - parallelization of applications
 - implementation of fault tolerance mechanisms



Key topics

Major advances

- Major theoretical advances in the development of operating systems:
 - The concept of process
 - Memory management
 - Resource scheduling and management
 - Information protection and security
- Each advance is characterized by principles, or abstractions, developed to meet difficult practical problems

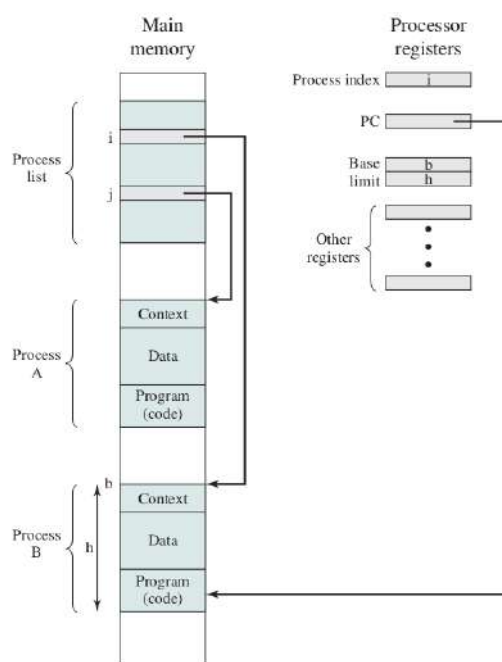
Key topics

The process

- It appears as a mean to control the activity of the various programs executing in a multiprogrammed system, possibly belonging to different users
 - A process **is not** a program
- **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:
 - **addressing space** – code and data (actual values of the different variables) of the associated program
 - input and output channels and data (data that are being transferred from input devices and to output devices)
 - actual values of the processor internal registers
 - state of execution
 - other process specific data (process ID, ...)
- Different processes can be running the same program

Key topics

A possible implementation of a process



- each process occupies a block of memory
 - containing the program, the data, and part of the context
- each process occupies a record in a process list
 - with a pointer to the process' memory
 - and the other part of the context
- a process index identifies the running process
- the base and limit registers define the running process memory
- the program counter and all data references are checked against these register values, at every access

Key topics

Memory management

- **Memory management** is required in order to support a flexible use of data
- Some OS memory management functionalities:
 - **Process isolation** – independent processes can not interfere with each other's memory
 - **Automatic allocation and management** – Programs should be dynamically allocated across the available memory
 - **Dynamic growth of used memory** – Memory used by programs should not be defined at start
 - **Protection and access control** – Sharing of memory should be possible, in a controlled way
 - **Long-term storage** – Many application programs need to store information for extended periods of time, after the computer has been powered down.
- This is accomplished with **virtual memory** and **file systems**
 - Virtual memory dissociates the memory see by a process and the real memory
 - File systems introduced the concept of file as the mean for long-term storage

Key topics

Resource scheduling and management

- There is a variety of resources shared among processes
 - main memory, I/O devices, processors, ...
- The OS must:
 - manage these resources
 - schedule their use by the various active processes
- The resource allocation and scheduling policy must provide:
 - **fairness** – give approximately equal and fair access to resources
 - **differential responsiveness** – act considering the total set of requirements
 - **efficiency** – attempt to maximize throughput, minimize response time, and, in the case of time sharing systems, accommodate as many users as possible
 - may not be possible, as conflicts may exist

Key topics

Information protection and security

- In time sharing systems, short-term and long-term data may belong to different users
 - thus, protection and security is a requirement
 - meaning controlling access to computer systems and to the information stored in them
- Points covered by OS:
 - **Availability** – protect the system against interruption
 - **Confidentiality** – ensure users cannot access unauthorized data
 - **Data integrity** – protect data from unauthorized modification
 - **Authenticity** – verify the identity of users and the validity of messages or data



Sistemas de Operação / Fundamentos de Sistemas Operativos

Semaphores and shared memory

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

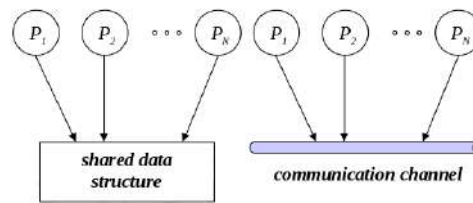
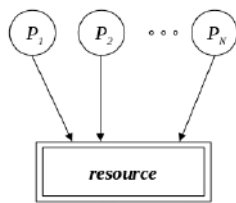
Outline

- ① Concepts
- ② Semaphores
- ③ Shared memory
- ④ Unix IPC primitives

Concepts

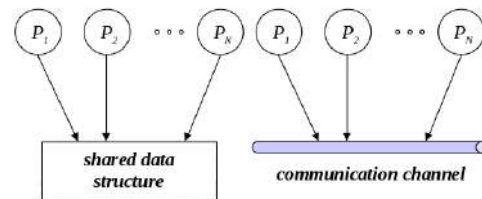
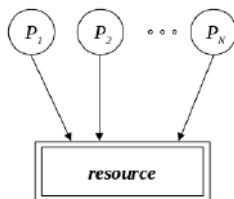
Independent and collaborative processes

- In a multiprogrammed environment, two or more processes can be:
 - **independent** – if they, from their creation to their termination, never explicitly interact
 - actually, there is an implicit interaction, as they compete for system resources
 - ex: jobs in a batch system; processes from different users
 - **cooperative** – if they share information or explicitly communicate
 - the **sharing** requires a **common address space**
 - **communication** can be done through a common address space or a **communication channel** connecting them



Concepts

Independent and collaborative processes (2)



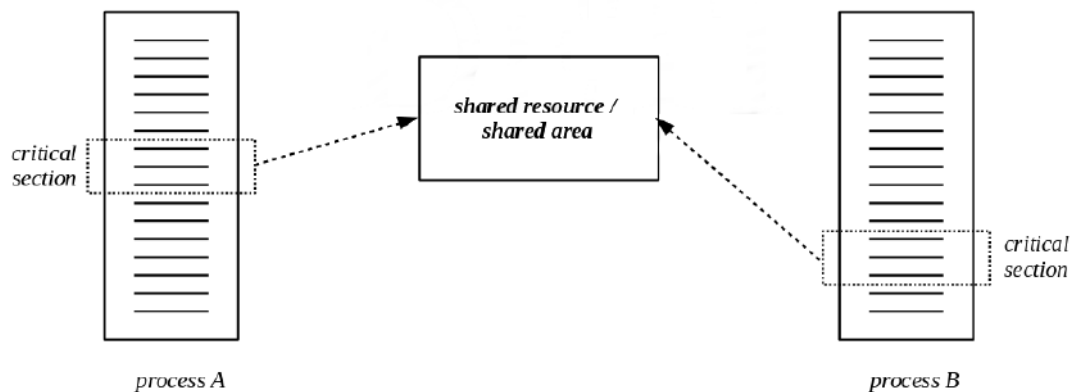
- **Independent processes** competing for a resource
- It is the **responsibility of the OS** to ensure the assignment of resources to processes is done in a controlled way, such that no information lost occurs
- In general, this imposes that only one process can use the resource at a time – **mutual exclusive access**
- The communication channel is typically a system resource, so processes compete for it

- **Cooperative processes** sharing information or communicating
- It is the **responsibility of the processes** to ensure that access to the shared area is done in a controlled way, such that no information lost occurs
- In general, this imposes that only one process can access the shared area at a time – **mutual exclusive access**
- The communication channel is typically a system resource, so processes compete for it

Concepts

Critical section

- Having access to a resource or to a shared area actually means **executing the code** that does the access
- This section of code, called **critical section**, if not properly protected, can result in **race conditions**
- A **race condition** is a condition where the behaviour (output, result) depends on the sequence or timing of other (uncontrollable) events
 - Can result in undesirable behaviour
- **Critical sections** should execute in **mutual exclusion**



Concepts

Deadlock and starvation

- Mutual exclusion in the access to a resource or shared area can result in
 - **deadlock** – when two or more processes are waiting forever to access to their respective critical section, waiting for events that can be demonstrated will never happen
 - operations are blocked
 - **starvation** – when one or more processes compete for access to a critical section and, due to a conjunction of circumstances in which new processes that exceed them continually arise, access is successively deferred
 - operations are continuously postponed

Semaphores

Definition

- A **semaphore** is a synchronization mechanism, defined by a data type plus two atomic operations, **down** and **up**
- Data type:

```
typedef struct
{
    unsigned int val;      /* can not be negative */
    PROCESS *queue;       /* queue of waiting blocked processes */
} SEMAPHORE;
```

- Operations:
 - **down**
 - block process if `val` is zero
 - decrement `val` otherwise
 - **up**
 - increment `val`
 - if `queue` is not empty, wake up one waiting process (accordingly to a given policy)
- Note that `val` can only be manipulated through these operations
 - It is not possible to check the value of `val`

Semaphores

An implementation of semaphores

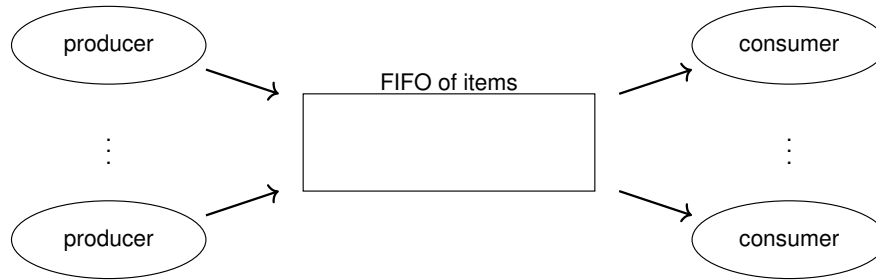
```
/* array of semaphores defined in kernel */
#define R ... /* semid = 0, 1, ..., R-1 */
static SEMAPHORE sem[R];
void sem_down(unsigned int semid)
{
    disable_interruptions;
    if (sem[semid].val == 0)
        block_on_sem(getpid(), semid);
    sem[semid].val -= 1;
    enable_interruptions;
}
void sem_up(unsigned int semid)
{
    disable_interruptions;
    sem[semid].val += 1;
    if (sem[semid].queue != NULL)
        wake_up_one_on_sem(semid);
    enable_interruptions;
}
```

- This implementation is typical of uniprocessor systems. Why?

- Semaphores can be binary or not binary
- How to implement **mutual exclusion** using semaphores?
 - Using a **binary** semaphore

Semaphores

Bounded-buffer problem – problem statement



- In this problem, a number of entities (producers) produce information that is consumed by a number of different entities (consumers)
- Communication is carried out through a buffer with bounded capacity
- Assume that every producer and every consumer run in a different process
 - Hence the FIFO must be implemented in **shared memory** so the different processes can access it
- How to guarantee that **race conditions** doesn't arise?
 - Enforcing **mutual exclusion** in the access to the FIFO

Semaphores

Bounded-buffer problem – implementation

```
shared FIFO fifo; /* fixed-size FIFO memory */
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        bool done = false;
        do
        {
            if (fifo.notFull())
            {
                fifo.insert(data);
                done = true;
            }
        } while (!done);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        bool done = false;
        do
        {
            if (fifo.notEmpty())
            {
                fifo.retrieve(&data);
                done = true;
            }
        } while (!done);
        consume_data(data);
        do_something_else();
    }
}
```

- This solution can suffer **race conditions**
 - How to avoid it?

Semaphores

Bounded-buffer problem – implementation

```
shared FIFO fifo; /* fixed-size FIFO memory */
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        bool done = false;
        do
        {
            if (fifo.notFull())
            {
                lock(p);
                fifo.insert(data);
                done = true;
                unlock(p);
            }
        } while (!done);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        bool done = false;
        do
        {
            if (fifo.notEmpty())
            {
                lock(c);
                fifo.retrieve(&data);
                done = true;
                unlock(c);
            }
        } while (!done);
        consume_data(data);
        do_something_else();
    }
}
```

- Introducing **mutual exclusion**
- **Mutual exclusion** is guaranteed, but suffers from **busy waiting**

Semaphores

Bounded-buffer problem – implementation

```
shared FIFO fifo; /* fixed-size FIFO memory */
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        bool done = false;
        do
        {
            if (fifo.notFull())
            {
                fifo.insert(data);
                done = true;
            }
        } while (!done);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        bool done = false;
        do
        {
            if (fifo.notEmpty())
            {
                fifo.retrieve(&data);
                done = true;
            }
        } while (!done);
        consume_data(data);
        do_something_else();
    }
}
```

- How to implement a safe solution using **semaphores**?
- guaranteeing **mutual exclusion** and absence of **busy waiting**

Semaphores

Bounded-buffer problem – solving using semaphores

```
shared FIFO fifo;      /* fixed-size FIFO memory */
shared sem access;     /* semaphore to control mutual exclusion */
shared sem nslots;     /* semaphore to control number of available slots */
shared sem nitems;     /* semaphore to control number of available items */

/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA val;

    forever
    {
        produce_data(&val);
        sem_down(nslots);
        sem_down(access);
        fifo.insert(val);
        sem_up(access);
        sem_up(nitems);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA val;

    forever
    {
        sem_down(nitems);
        sem_down(access);
        fifo.retrieve(&val);
        sem_up(access);
        sem_up(nslots);
        consume_data(val);
        do_something_else();
    }
}
```

- `fifo.notEmpty()` and `fifo.notFull()` are no longer necessary. Why?
- What are the initial values of the semaphores?

Semaphores

Bounded-buffer problem – wrong solution

```
shared FIFO fifo;      /* fixed-size FIFO memory */
shared sem access;     /* semaphore to control mutual exclusion */
shared sem nslots;     /* semaphore to control number of available slots */
shared sem nitems;     /* semaphore to control number of available items */

/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA val;

    forever
    {
        produce_data(&val);
        sem_down(access);
        sem_down(nslots);
        fifo.insert(val);
        sem_up(access);
        sem_up(nitems);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA val;

    forever
    {
        sem_down(nitems);
        sem_down(access);
        fifo.retrieve(&val);
        sem_up(access);
        sem_up(nslots);
        consume_data(val);
        do_something_else();
    }
}
```

- One can easily make a mistake
 - What is **wrong** with this solution? It can cause deadlock

Semaphores

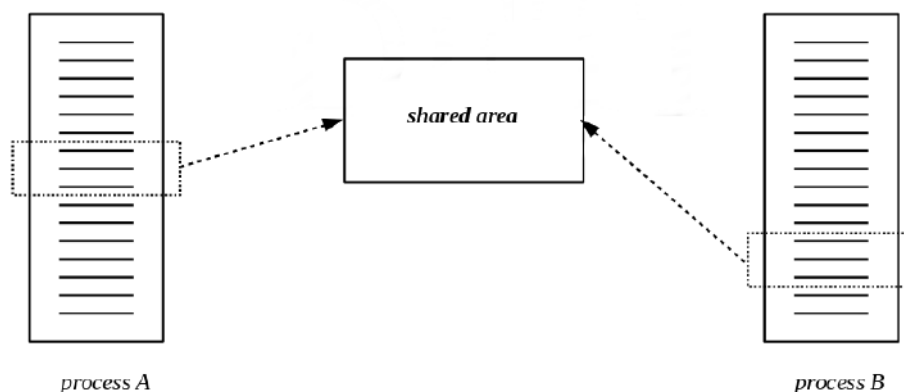
Analysis of semaphores

- Concurrent solutions based on semaphores have advantages and disadvantages
- **Advantages:**
 - **support at the operating system level**– operations on semaphores are implemented by the kernel and made available to programmers as system calls
 - **general**– they are low level constructions and so they are versatile, being able to be used in any type of solution
- **Disadvantages:**
 - **specialized knowledge**– the programmer must be aware of concurrent programming principles, as race conditions or deadlock can be easily introduced
 - See the previous example, as an illustration of this

Shared memory

A resource

- Address spaces of processes are independent
- But address spaces are virtual
- The same physical region can be mapped into two or more virtual regions
- **Shared memory** is managed as a resource by the operating system
- Two actions are required:
 - Requesting a segment of shared memory to the OS
 - Mapping that segment in the process' address space



Unix IPC primitives

Semaphores

- **System V semaphores**
 - creation: `semget`
 - down and up: `semop`
 - other operations: `semctl`
 - execute `man semget`, `man semop` or `man semctl` for a description
- **POSIX semaphores**
 - Two types: named and unnamed semaphores
 - Named semaphores
 - `sem_open`, `sem_close`, `sem_unlink`
 - created in a virtual filesystem (e.g., `/dev/sem`)
 - unnamed semaphores – memory based
 - `sem_init`, `sem_destroy`
 - down and up
 - `sem_wait`, `sem_trywait`, `sem_timedwait`, `sem_post`
 - execute `man sem_overview` for an overview

Unix IPC primitives

Shared memory

- **System V shared memory**
 - creation – `shmget`
 - mapping and unmapping – `shmat`, `shmdt`
 - other operations – `shmctl`
 - execute `man shmget`, `man shmat`, `man shmdt` or `man shmctl` for a description
- **POSIX shared memory**
 - creation - `shm_open`, `ftruncate`
 - mapping and unmapping - `mmap`, `munmap`
 - other operations - `close`, `shm_unlink`, `fchmod`, ...
 - execute `man shm_overview` for an overview



Sistemas de Operação / Fundamentos de Sistemas Operativos

Threads, mutexes and condition variables in Unix/Linux

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

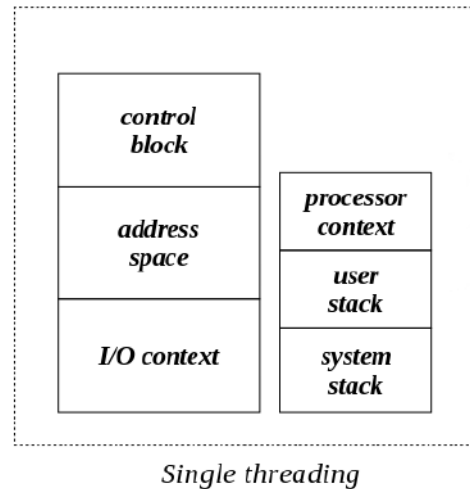
Outline

- ① Threads and multithreading
- ② Threads in Linux
- ③ Monitors
- ④ POSIX support for monitors

Threads

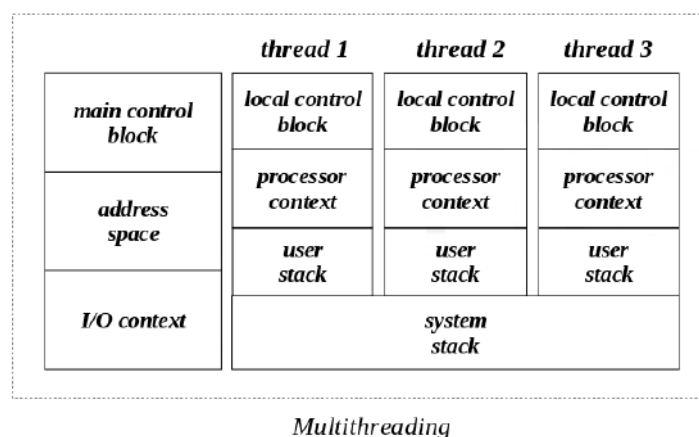
Single threading

- In traditional operating system, a process includes:
 - an address space (code and data of the associated program)
 - 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 separately
- In this model, **thread** appears as an execution component within a process



Threads

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**
- Threads can be seen as **light weight processes**

Threads

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 involves substantial I/O, multithreading allows activities to overlap, thus speeding up its execution
- **multiprocessing** – real parallelism is possible if multiples CPUs exist

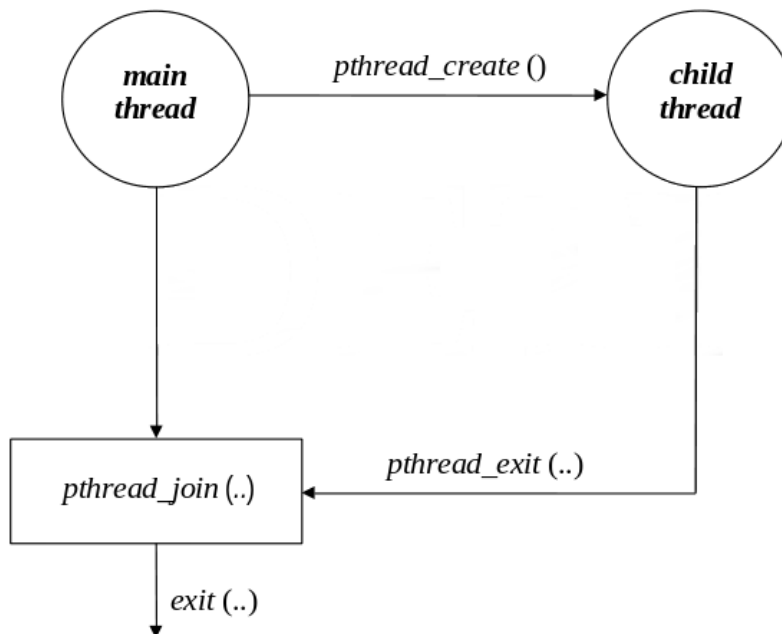
Threads in linux

The `clone` system call

- 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
 - the child starts execution in the point of the forking
 - **clone** – creates a new process that can share elements with its parent
 - address space, table of file descriptors, and table of signal handlers are shareable.
 - the child starts execution in a specified 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

Thread creation and termination – pthread library



Threads in linux

Thread creation and termination – example

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

/* return status */
int status;

/* child thread */
void *threadChild (void *par)
{
    printf ("I'm the child thread!\n");
    sleep(1);
    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;
}
```

Monitors

Introduction

- A problem with semaphores is that they are used both to implement **mutual exclusion** and to **synchronize** processes
 - Being low level primitives, they are applied in a **bottom-up** perspective
 - if required conditions are not satisfied, processes are blocked before they enter their critical sections
 - this approach is prone to errors, mainly in complex situations, as synchronization points can be scattered throughout the program
 - A higher level approach should followed a **top-down** perspective
 - processes must first enter their critical sections and then block if continuation conditions are not satisfied
 - A solution is to introduce a (concurrent) construction at the programming level that deals with mutual exclusion and synchronization separately
-
- A **monitor** is a synchronization mechanism, independently proposed by Hoare and Brinch Hansen, supported by a (concurrent) programming language
 - The **pthread** library provides primitives that allows to implement monitors (of the Lampson-Redell type)

Monitors

Definition

```
monitor example
{
    /* internal shared data structure */
    DATA data;

    cond c; /* condition variable */

    /* access methods */
    method_1 (...)
    {
        ...
    }

    method_2 (...)
    {
        ...
    }

    ...

    /* initialization code */
    ...
}
```

- An application is seen as a set of threads that compete to access the **shared data** structure
- This shared data can only be accessed through the access methods
- Every method is executed in **mutual exclusion**
- If a thread calls an access method while another thread is inside another access method, its execution is blocked until the other leaves
- Synchronization between threads is possible through **condition variables**
- Two operation on them are possible:
 - **wait** – the thread is blocked and put outside the monitor
 - **signal** – if there are threads blocked, one is waked up. *Which one?*

Monitors

Bounded-buffer problem – solving using monitors

```
shared FIFO fifo; /* fixed-size FIFO memory */
shared mutex access; /* mutex to control mutual exclusion */
shared cond nslots; /* condition variable to control availability of slots */
shared cond nitems; /* condition variable to control availability of items */

/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        lock(access);
        if/while (fifo.isFull())
        {
            wait(nslots, access);
        }
        fifo.insert(data);
        signal(nitems);
        unlock(access);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        lock(access);
        if/while (fifo.isEmpty())
        {
            wait(nitems, access);
        }
        fifo.retrieve(&data);
        signal(nslots);
        unlock(access);
        consume_data(data);
        do_something_else();
    }
}
```

- The **mutex** is the resource used to control mutual exclusion
- **Critical sections** are explicitly framed by the **lock** and **unlock** of a mutex

Unix IPC primitives

POSIX support for monitors

- Standard POSIX, IEEE 1003.1c, defines a programming interface (API) for the creation and synchronization of threads
 - In unix, this interface is implemented by the **pthread** library
- It allows for the implementation of monitors in C/C++
 - Using mutexes and condition variables
 - Note that they are of the **Lampson / Redell** type
- Some of the available functions:
 - **pthread_create** – creates a new thread; similar to **fork**
 - **pthread_exit** – equivalent to **exit**
 - **pthread_join** – equivalent a **waitpid**
 - **pthread_self** – equivalent a **getpid()**
 - **pthread_mutex_*** – manipulation of mutexes
 - **pthread_cond_*** – manipulation of condition variables
 - **pthread_once** – initialization



Sistemas de Operação / Fundamentos de Sistemas Operativos

Processes

Artur Pereira <artur@ua.pt>

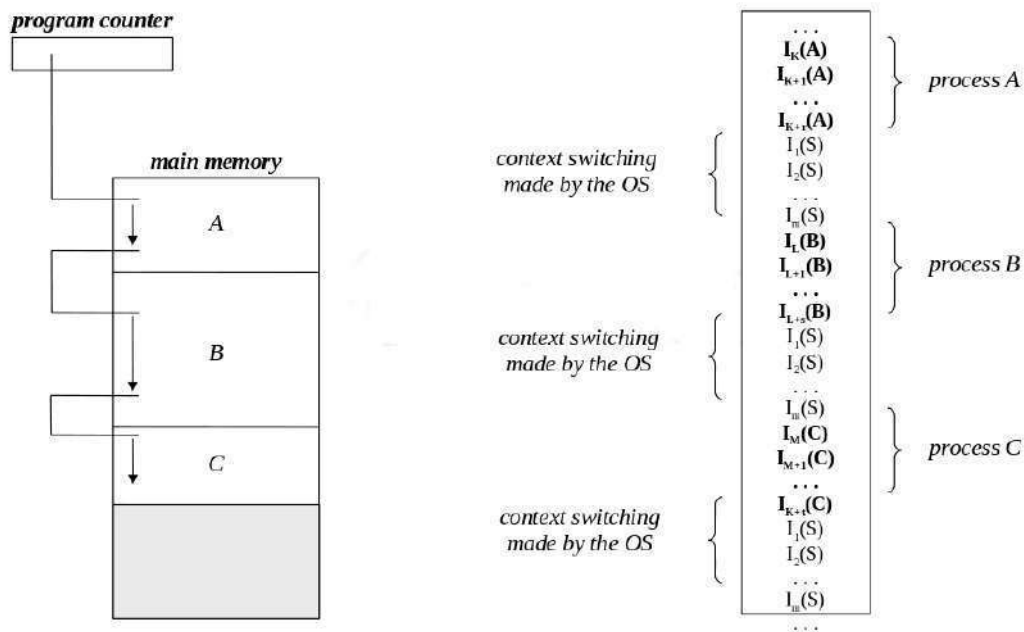
DETI / Universidade de Aveiro

Outline

- ① Process model
- ② Process state diagram
- ③ Process control table
- ④ Context switching
- ⑤ Threads and multithreading
- ⑥ Bibliography

Process

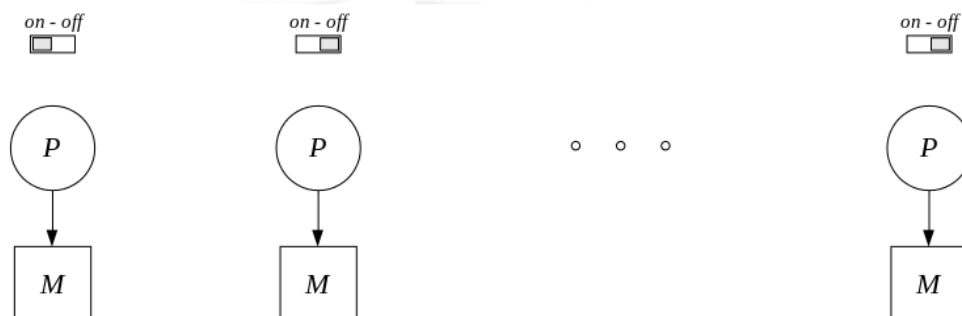
Execution in a multiprogrammed environment



Processes

Process model

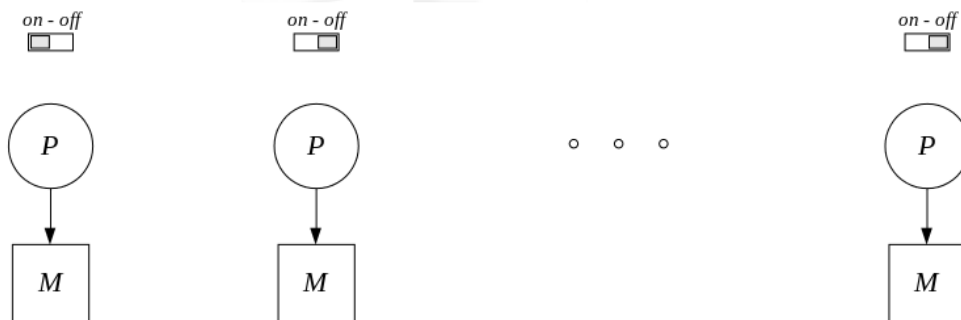
- In **multiprogramming** the activity of the processor, because it is switching back and forth from process to process, is hard to perceive
- Thus, it is better to assume the existence of a number of virtual processors, one per existing process
 - Turning off one virtual processor and on another, corresponds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



Processes

Process model (2)

- The switching between processes, and thus the switching between virtual processors, can occur for different reasons, possibly 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



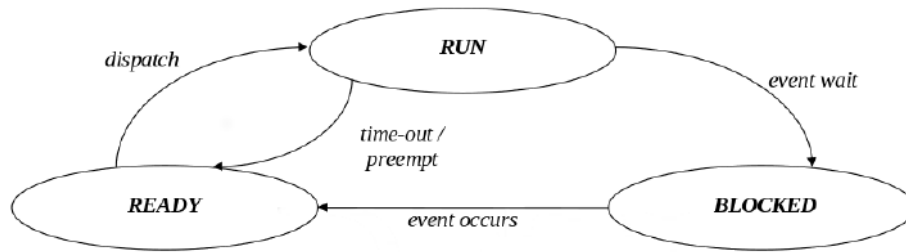
Processes

(Short-term) Process states

- A process can be **not running** for different reasons
 - so, one should identify the possible process **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, in some cases, can 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
 - They will be covered later

Processes

Short-term state diagram



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

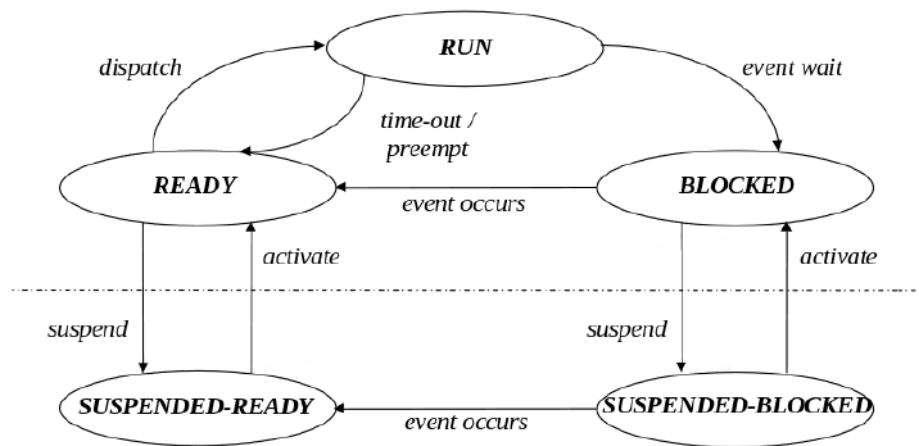
Processes

Medium-term states

- 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

Processes

State diagram, including short- and medium-term states



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

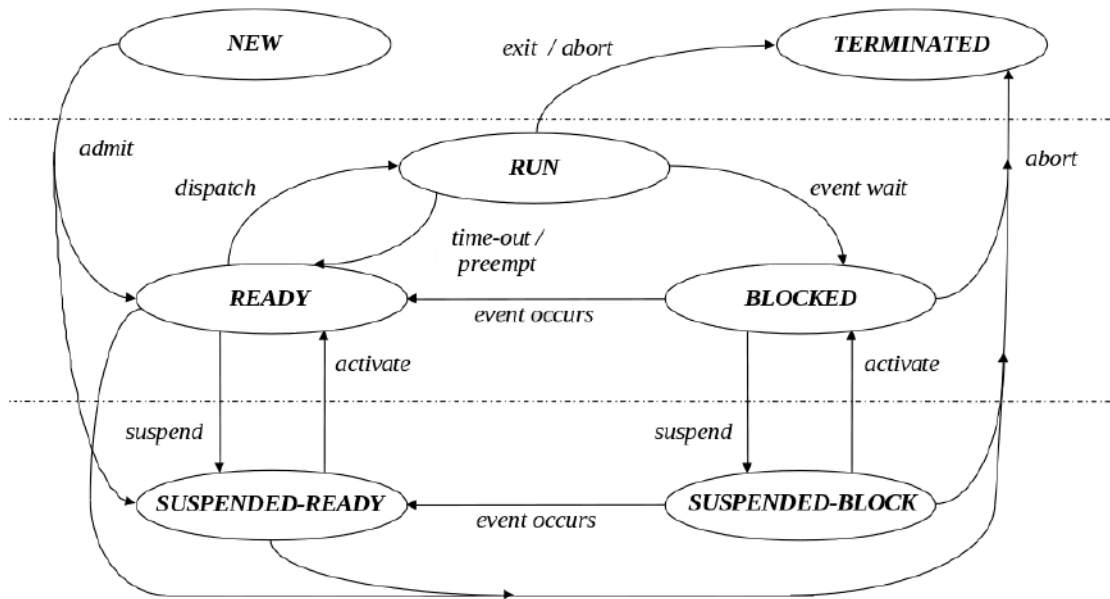
Processes

Long-term states and transitions

- 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 terminate
- 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, but 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)

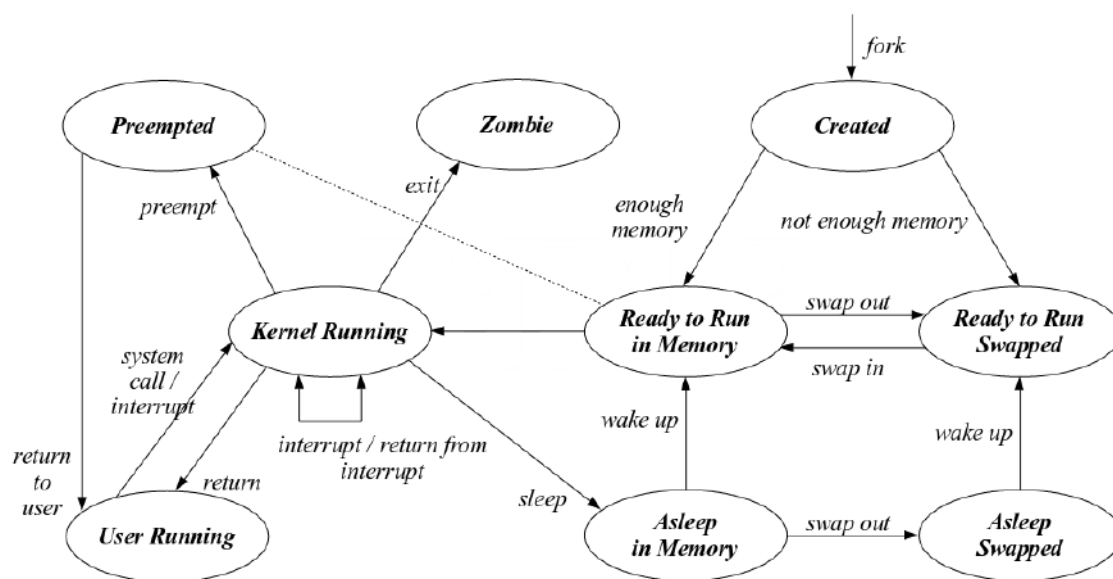
Processes

Global state diagram



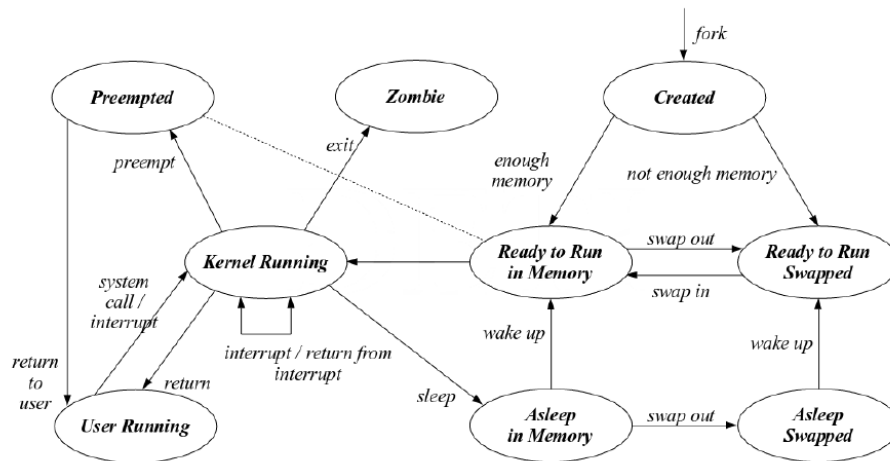
Processes

Typical Unix state diagram



Processes

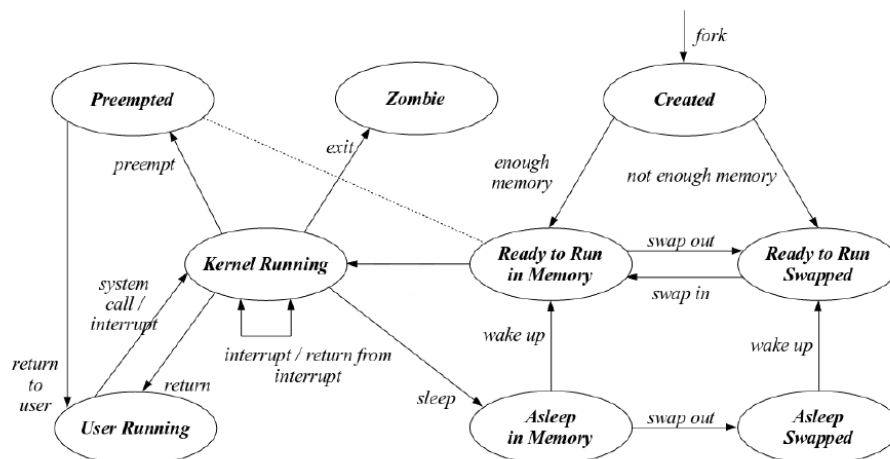
Typical Unix state diagram (2)



- 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, represented by the dashed line

Processes

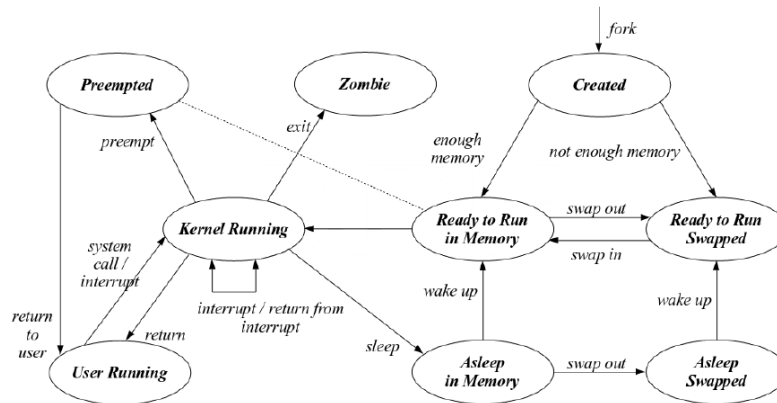
Typical Unix state diagram (3)



- 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

Processes

Typical Unix state diagram (4)

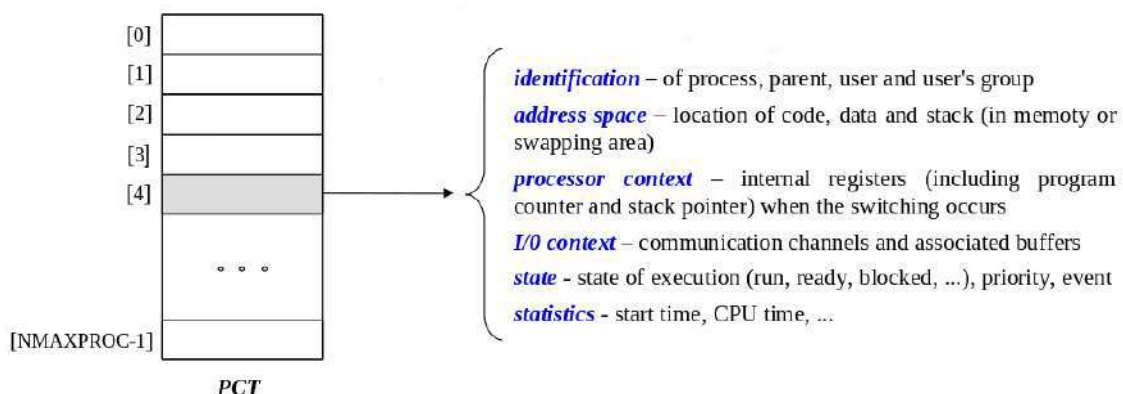


- Traditionally, execution in supervisor mode could not be interrupted (thus UNIX does not allow real time processing)
- In current versions, namely from SVR4, the problem was solved by dividing the code into a succession of atomic regions between which the internal data structures are in a safe state 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**

Processes

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)**, which can be seen as an array of process control blocks, stores information about all processes



Processes

Context switching

- Current processors have two functioning modes:
 - **supervisor mode** – all instruction set can be executed
 - is a privileged mode
 - **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
 - external to the execution of the current instruction
 - illegal instruction (division by zero, bus error)
 - associated with the execution of the current instruction, but not intended
 - trap instruction (software interruption)
 - associated with the execution of the current instruction, and intended

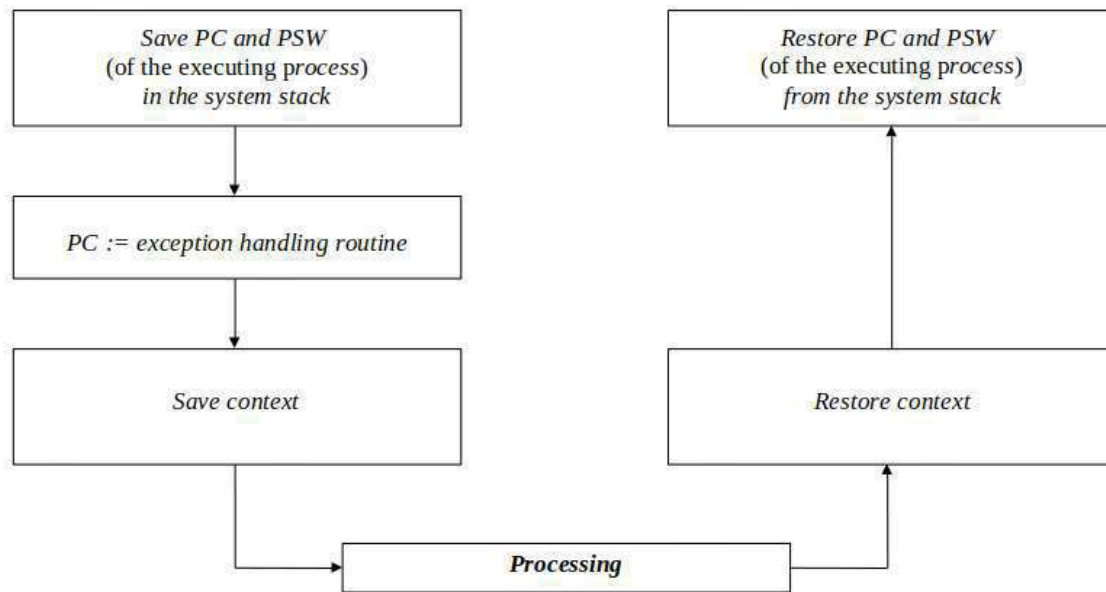
Processes

Context switching (2)

- 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)
 - This establishes a uniform operating environment: **exception handling**
-
- **Context switching** is the process of storing the state of a process and restoring the state of another process
 - Context switching occurs necessarily in the context of an exception, with a small difference on how it is handle

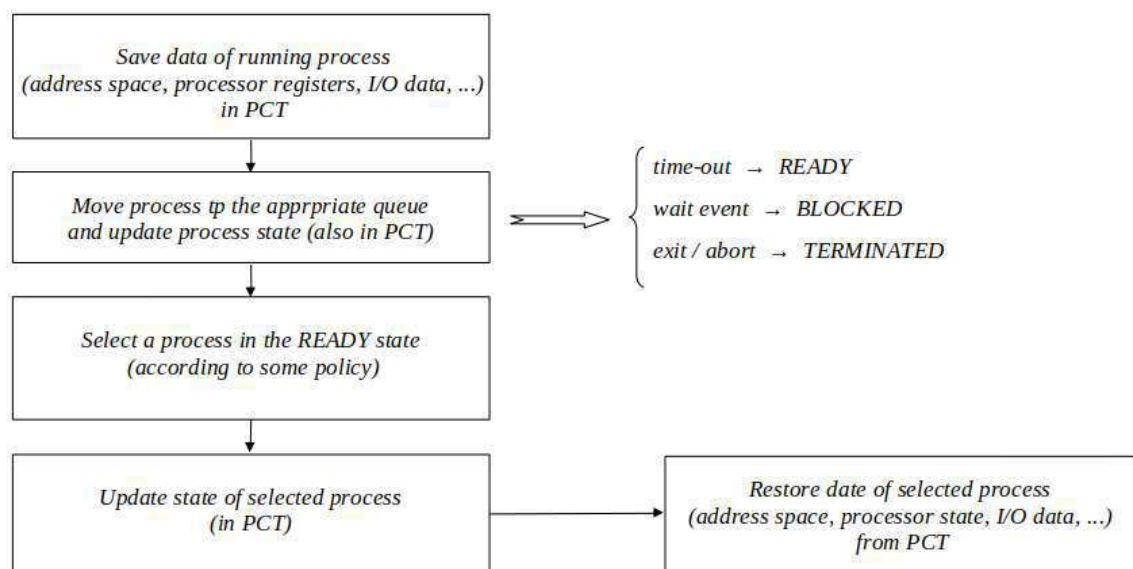
Process

Processing a (normal) exception



Process

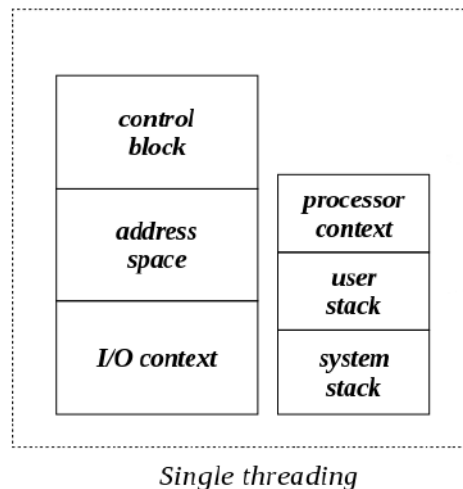
Processing a process switching



Threads

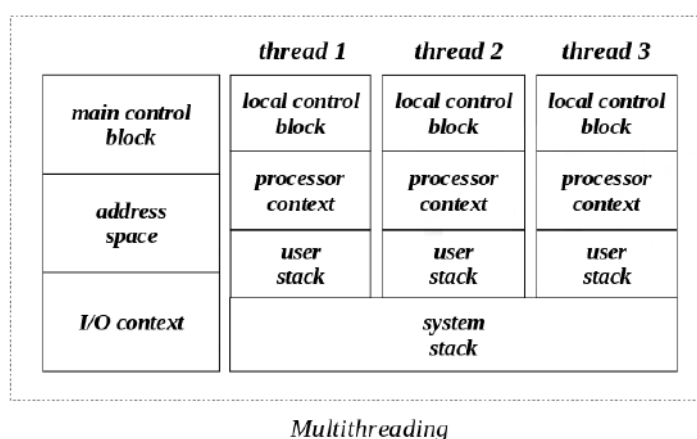
Single threading

- In traditional operating system, a process includes:
 - an address space (code and data of the associated program)
 - 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 separately
- In this model, **thread** appears as an execution component within a process



Threads

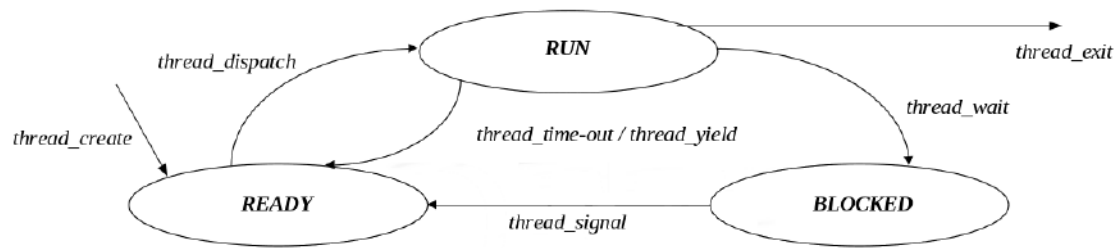
Multithreading



- 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**
- Threads can be seen as **light weight processes**

Threads

State diagram of a thread



- Only states concerning the management of the processor are considered (short-term states)
- states **suspended-ready** and **suspended-blocked** are not present:
 - they are related to the process, not to the threads
- states **new** and **terminated** are not present:
 - the management of the multiprogramming environment is basically related to restrict the number of threads that can exist within a process

Bibliography

- Operating Systems: Internals and Design Principles, W. Stallings, Prentice-Hall International Editions, 7th Ed, 2012
 - 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)
- Operating Systems Concepts, A. Silberschatz, P. Galvin and G. Gagne, John Wiley & Sons, 9th Ed, 2013
 - Chapter 3: Processes (sections 3.1 to 3.3)
 - Chapter 4: Threads (sections 4.1 and 4.4.1)
- Modern Operating Systems, A. Tanenbaum and H. Bos, Pearson Education Limited, 4th Ed, 2015
 - Chapter 2: Processes and Threads (sections 2.1 and 2.2)



Sistemas de Operação / Fundamentos de Sistemas Operativos

Processor scheduling

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

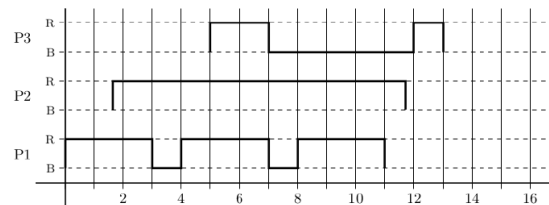
Outline

- 1 Introduction
- 2 Evaluating a scheduling algorithm
- 3 Priorities
- 4 Scheduling algorithms
- 5 Scheduling in Linux
- 6 Bibliography

Processor scheduler

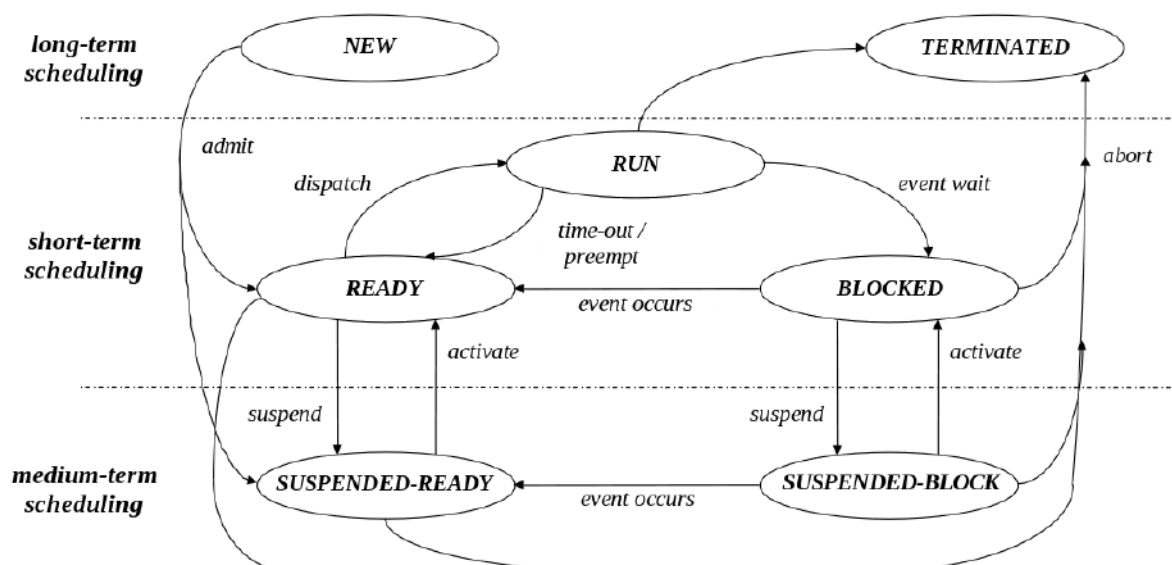
Definition

- When seen by its own, the execution of a process is an alternate sequence of two type of periods:
 - **CPU burst** – executing CPU instructions
 - **I/O burst** – waiting for the completion of an I/O request
- Based on this, a process can be classified as:
 - **I/O-bound** – if it has many short CPU bursts
 - **CPU-bound** (or **processor-bound**) – if it has (few) long CPU bursts
- The idea behind multiprogramming is to take advantage of the I/O burst periods to put other processes using the CPU
- The **processor scheduler** is the component of the operating system responsible for deciding how the CPU is assigned for the execution of the CPU bursts of the several processes that coexist in the system



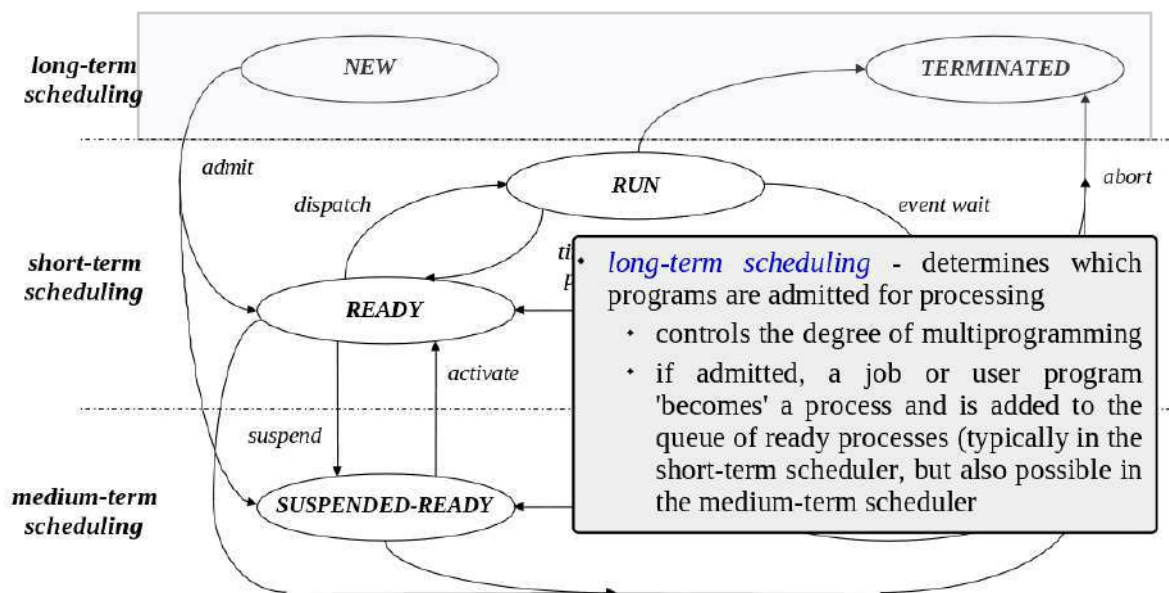
Processor scheduler

Levels in processor scheduling



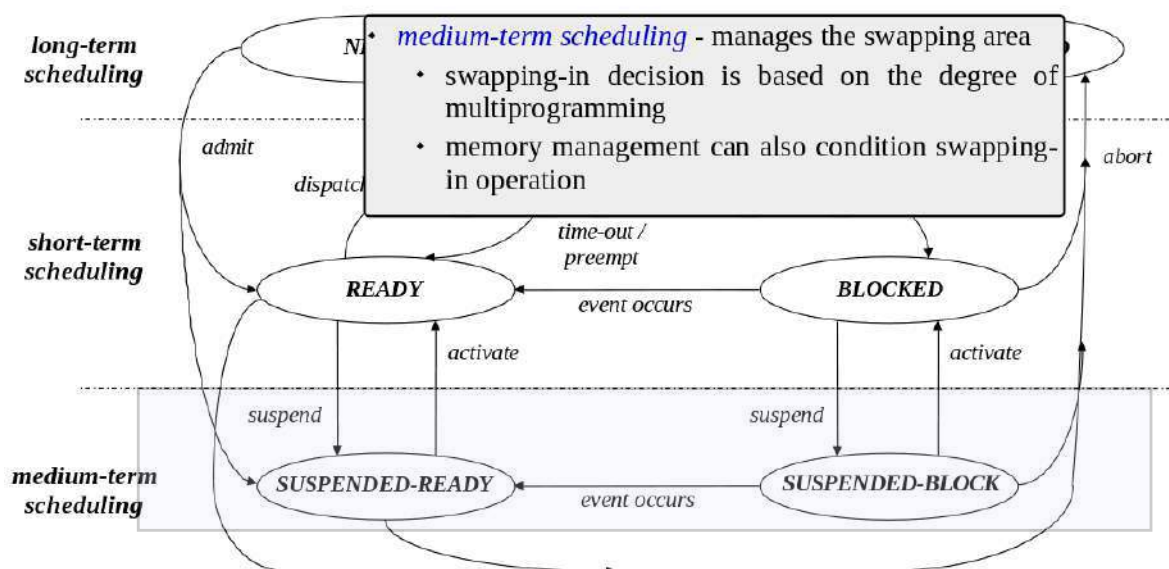
Processor scheduler

Long-term scheduling



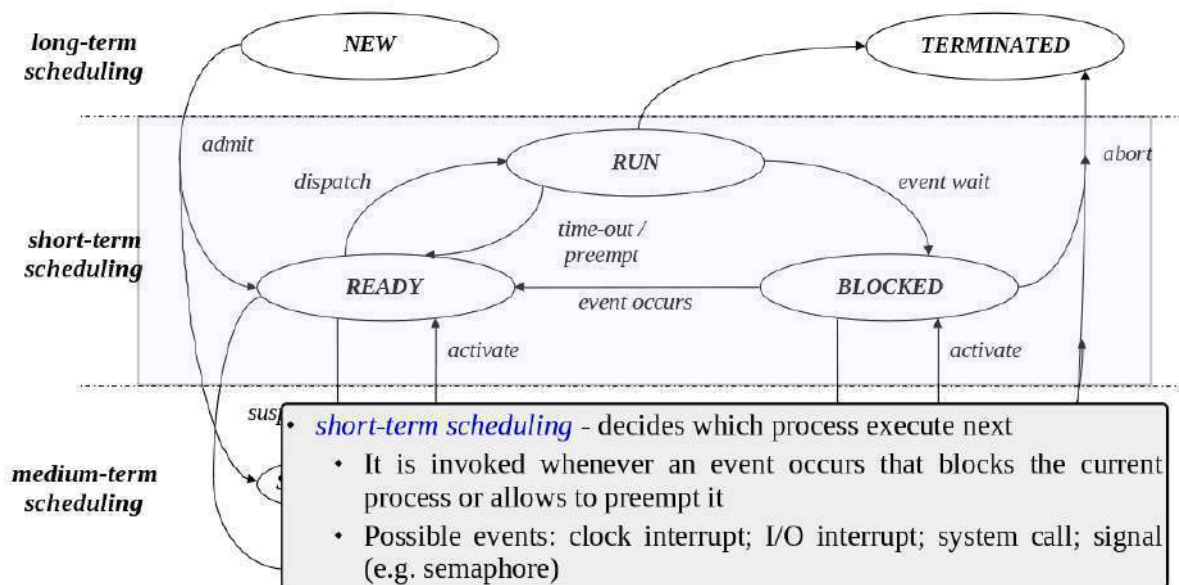
Processor scheduler

Medium-term scheduling



Processor scheduler

Short-term scheduling



Short-term processor scheduler

Preemption & non-preemption

- The short-term processor scheduler can be **preemptive** or **non-preemptive**
- **Non-preemptive scheduling** – a process keeps the processor until it blocks or ends
 - Transitions **time-out** and **preempt** do not exist
 - Typical in batch systems. *Why?*
- **Preemptive scheduling** – a process can lose the processor due to external reasons
 - by exhaustion of the assigned time quantum (**time-out** transition)
 - because a more priority process becomes ready to run (**preempt** transition)
- Interactive systems must be preemptive. *Why?*
- What type to use in a real-time system? *Why?*

Scheduling algorithms

Evaluation criteria

- The main objective of short-term scheduling is to allocate processor time in order to optimize some objective function of the system behavior
- A set of criteria must be established for the evaluation
- These criteria can be seen from different perspectives:
 - **User-oriented** criteria – related to the behavior of the system as perceived by the individual user or process
 - **System-oriented** criteria – related to the effective and efficient utilization of the processor
 - A criterion can be a direct/indirect **quantitative** measure or just a **qualitative** evaluation
- Scheduling criteria are **interdependent**, thus it is impossible to optimize all of them simultaneously

Scheduling criteria

User-oriented scheduling criteria

- **Turnaround time** – interval of time between the submission of a process/job and its completion (includes actual execution time plus time spent waiting for resources, including the processor)
 - appropriate measure for a batch job
 - should be minimized
- **Waiting time** – sum of periods spent by a process waiting in the ready state
 - should be minimized
- **Response time** – time from the submission of a request until the response begins to be received
 - appropriate measure for an interactive process
 - should be minimized
 - but also the number of interactive processes with acceptable response time should also be maximized
- **Deadlines** – time of completion of a process
 - percentage of deadlines met should be maximized, even subordinating other goals
- **Predictability** – how response is affected by the load on the system
 - A given job should run in about the same amount of time and at about the same cost regardless of the load on the system

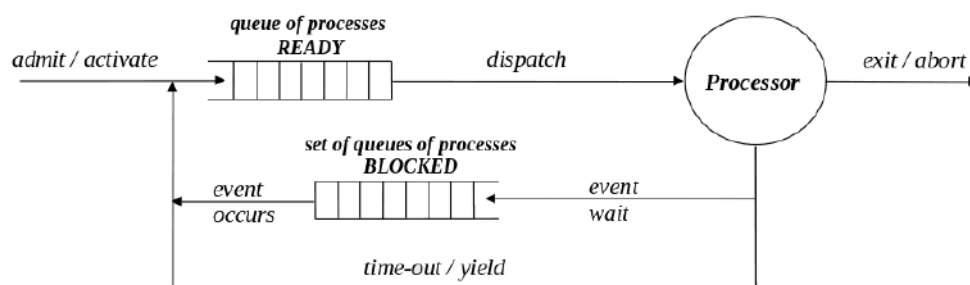
Scheduling criteria

System-oriented scheduling criteria

- **Fairness** – equality of treatment
 - In the absence of guidance, processes should be treated the same, and no process should suffer starvation
 - **Throughput** – number of processes completed per unit of time
 - measures the amount of work being performed by the system
 - should be maximized
 - depends on the average lengths of processes but also on the scheduling policy
 - **Processor utilization** – percentage of time that the processor is busy
 - should be maximized (specially in expensive shared systems)
 - **Enforcing priorities**
 - higher-priority processes should be favoured
- As referred to before, it is impossible to satisfy all criteria simultaneously
- Those to favour depends on application

Priorities

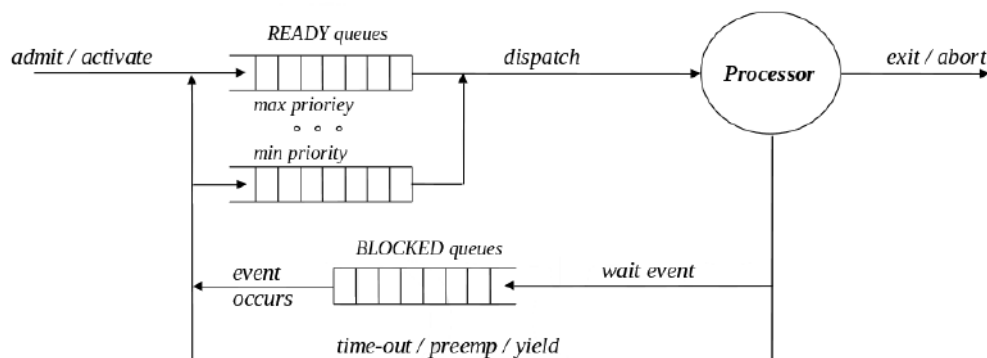
Without priorities, favouring fairness



- All processes are equal, being served in order of arrival to the READY queue
 - In non-preemptive scheduling, it is normally referred to as **first-come, first served (FCFS)** – the **time-out** transition does not exist
 - In preemptive scheduling, it is normally referred to as **round robin** – the **time-out** transition exist
- Easy to implement
- Favours CPU-bound processes over I/O-bound ones
- In interactive systems, the **time quantum** should be carefully chosen in order to get a good compromise between fairness and response time

Priorities

Enforcing priorities



- Often, being all the same is not the most appropriate
 - in interactive systems, minimizing response time requires I/O-bound processes to be privileged
 - in real-time systems, there are processes (for example, those associated with alarm events or operational actions) that have severe time constraints
- To address this, processes can be grouped in different levels of priority
 - higher-priority processes are selected first for execution
 - lower-priority processes can suffer **starvation**

Priorities

Types of priorities

- Priorities can be:
 - **static** – if they do not change over time
 - **deterministic** – if they are deterministically defined
 - **dynamic** – if they depend on the past history of execution of the processes
- **Static priorities:**
 - Processes are grouped into fixed priority classes, according to their relative importance
 - Clear risk of **starvation** of lower-priority processes
 - The most **unfair** discipline
 - Typical in real-time systems – **why?**
- **Deterministically changing priorities:**
 - When a process is created, a given priority level is assigned to it
 - on **time-out** the priority is decremented
 - on **event wait** the priority is incremented
 - when a minimum value is reached, the priority is set to the initial value

Priorities

Types of priorities (2)

- **Dynamic priorities:**

- Priority classes are functionally defined
 - In interactive systems, change of class can be based on how the last execution window was used
 - level 1 (highest priority): **terminals** – a process enters this class on event occurs if it was waiting for data from the standard input device
 - level 2: **generic I/O** – a process enters this class on event occurs if it was waiting for data from another type of input device
 - level 3: **small time quantum** – a process enters this class on time-out
 - level 4: (lowest priority): **large time quantum** – a process enters this class after a successive number of time-outs.
- They are clearly CPU-bound processes and the idea is given them large execution windows, less times

Priorities

Types of priorities (3)

- **Dynamic priorities:**

- In batch systems, the turnaround time should be minimized
- If estimates of the execution times of a set of processes are known in advance, it is possible to establish an order for the execution of the processes that minimizes the average turnaround time of the group.
- Assume N jobs are submitted at time 0, whose estimates of the execution times are t_{e_n} , with $n = 1, 2, \dots, N$.

- The turnaround time of job i is given by

$$t_{t_i} = t_{e_1} + t_{e_2} + \dots + t_{e_i}$$

- The average turnaround time is given by

$$t_m = \frac{1}{N} \sum_{i=1}^N t_{t_i} = t_{e_1} + \frac{N-1}{N} t_{e_2} + \dots + \frac{1}{N} t_{e_N}$$

- t_m is **minimum** if jobs are scheduled in ascending order of the estimated execution times

Priorities

Types of priorities (4)

- **Dynamic priorities:**

- An approach similar to the previous one can be used in interactive systems
- The idea is to estimate the occupancy fraction of the next execution window, based on the occupation of the past windows, and assign the processor to the process for which this estimate is the lowest
- Let e_1 be the estimate of the occupancy fraction of the first execution window assigned to a process and let f_1 be the first fraction effectively verified. Then:

- estimate e_2 is given by

$$e_2 = a.e_1 + (1 - a).f_1 \quad , \quad a \in [0, 1]$$

- estimate e_N is given by

$$\begin{aligned} e_N &= a.e_{N-1} + (1 - a).f_{N-1} \quad , \quad a \in [0, 1] \\ &= a^{N-1}.e_1 + a^{N-2}.(1 - a).f_1 + \dots + a.(1 - a).f_{N-2} + (1 - a).f_{N-1} \end{aligned}$$

- coefficient a is used to control how much the past history of execution influences the present estimate

Priorities

Types of priorities (5)

- In the previous approach, CPU-bound processes can suffer **starvation**
- To overcome that problem, the **aging** of a process in the READY queue can be part of the equation
- Let R be such time, typically normalized in terms of the duration of the execution interval.
- Then, priority p of a process can be given by

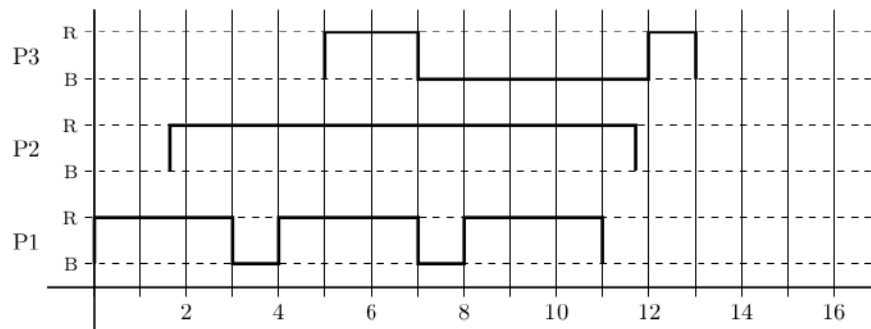
$$p = \frac{1 + b.R}{e_N}$$

where b is a coefficient that controls how much the aging weights in the priority

Scheduling policies

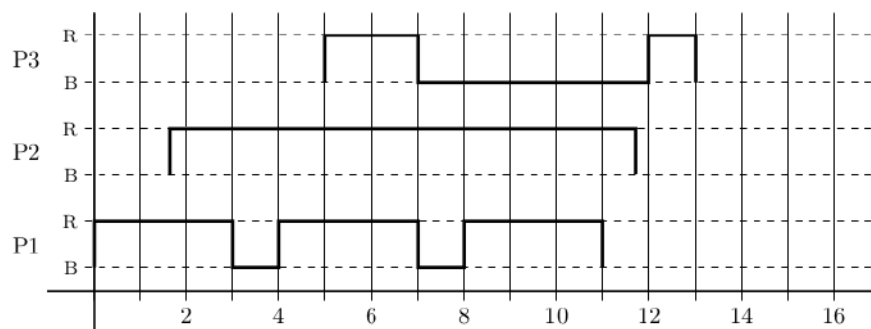
FCFS

- **First-Come-First-Server (FCFS) scheduler**
 - Also known as **First-In-First-Out (FIFO)**
 - The oldest process in the **READY** queue is the first to be selected
 - Non-preemptive (in strict sense)
 - Can be **combined with a priority schema** (in which case preemption exists)
 - Favours CPU-bound processes over I/O-bound
 - Can result in bad use of both processor and I/O devices

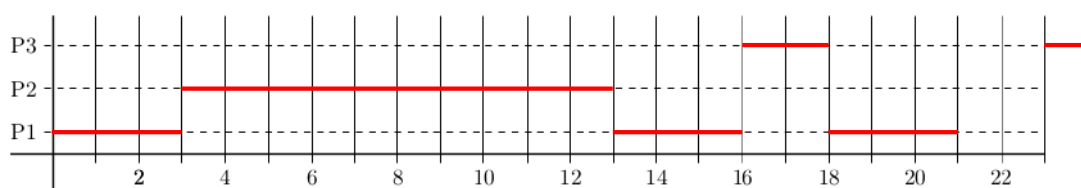


Scheduling policies

FCFS – example



- Draw processor utilization, assuming FCFS policy and no priorities



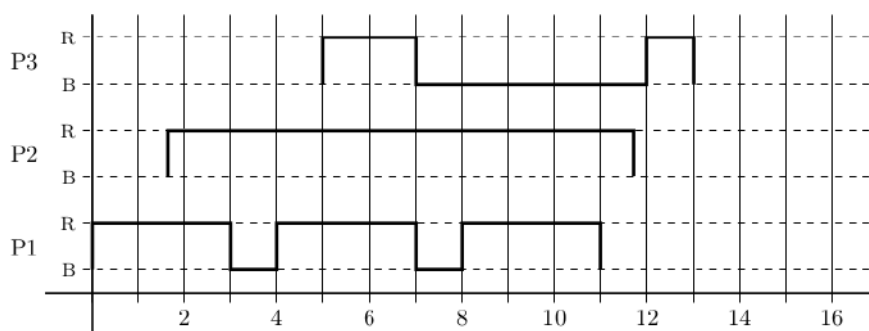
Scheduling policies

RR

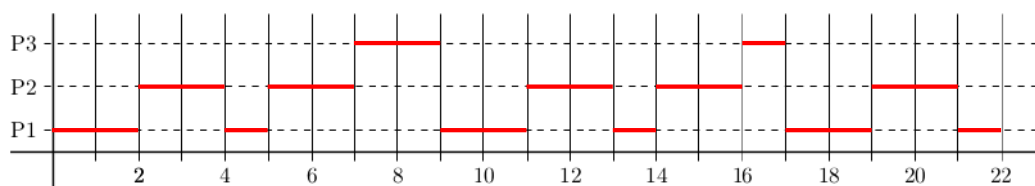
- Round robin (RR) scheduler
 - Preemptive (base on a clock)
 - Each process is given a maximum slice of time before being preempted (time quantum)
 - Also known as time slicing
 - The oldest process in the READY queue is the first one to be selected (no priorities)
 - Can be combined with a priority schema
 - The principal design issue is the time quantum
 - very short is good, because short processes will move through the system quickly and response time is minimized
 - very short is bad, because every context switching involves a processing overhead
 - Effective in general purpose time-sharing systems and in transaction processing systems
 - Favours CPU-bound processes over I/O-bound
 - Can result in bad use of I/O devices

Scheduling policies

FCFS – example



- Draw processor utilization, assuming RR policy with a time quantum of 2 and no priorities



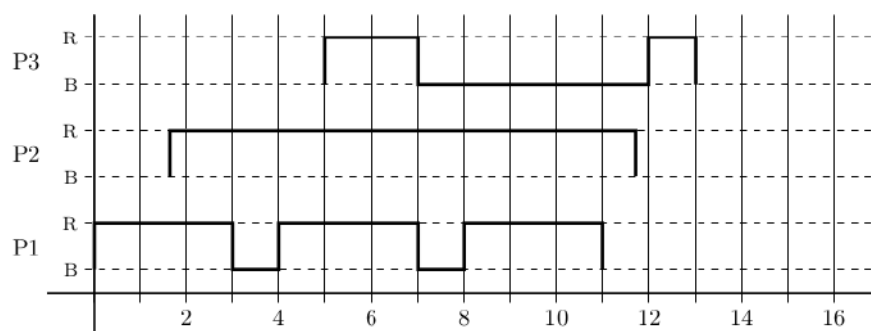
Scheduling policies

SPN

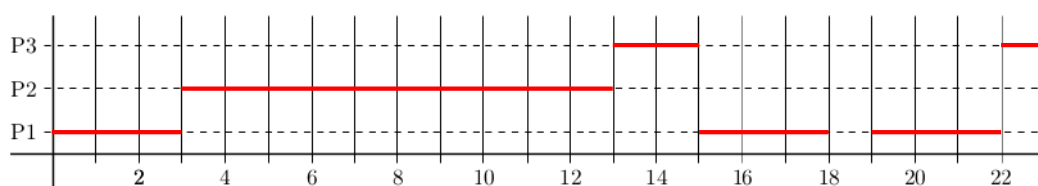
- Shortest Process Next (SPN) scheduler
 - Also known as **shortest job first (SJF)**
 - Non-preemptive
 - The process with the **shortest expected next CPU burst time** is selected next
 - FCFS is used to tie up (in case several processes have the same burst time)
 - Maximum throughput
 - Minimum average waiting time and turnaround time
 - Risk of **starvation** for longer processes
 - Requires the knowledge in advance of the **(expected) processing time**
 - This value can be predicted, using the previous values
 - Used in long-term scheduling in batch systems
 - where users are motivated to estimate the process time limit accurately

Scheduling policies

FCFS – example



- Draw processor utilization, assuming SPN policy and no priorities



Scheduling policies in Linux

Different classes

- Linux considers 3 scheduling classes, each with multiple priority levels:
 - **SCHED_FIFO** – FIFO real-time threads, with priorities
 - a running thread in this class is preempted only if a higher priority process (of the same class) becomes ready
 - a running thread can voluntarily give up the processor, executing primitive `sched_yield`
 - within the same priority an FCFS discipline is used
 - **SCHED_RR** – Round-robin real-time threads, with priorities
 - additionally, a running process in this class is preempted if its time quantum ends
 - **SCHED_OTHER** – non-real-time threads
 - can only execute if no real-time thread is ready to execute
 - associated to user processes
 - the scheduling police changed along kernel versions
- priorities range from 0 to 99 for real-time threads and 100 to 139 for the others
- `nice` system call allows to change the priority of non-real time threads

Scheduling policies in Linux

Traditional algorithm for the **SCHED_OTHER**

- In class **SCHED_OTHER**, priorities are based on credits
- Credits of the running process are decremented at every RTC interrupt
 - the process is preempted when **zero credits** are reached
- When all ready processes have zero credits, credits for all processes, including those that are blocked, are recalculated
 - Recalculation is done based on formula
$$\text{CPU}(i) = \frac{\text{CPU}(i-1)}{2} + \text{PBase} + \text{nice}$$
- Past history of execution and priorities are combined in the algorithm
- Response time of I/O-bound processes is minimized
- Starvation of processor-bound processes is avoided
- Not adequate for multiple processors and bad if the number of processes is high

Scheduling policies in Linux

New algorithm for the `SCHED_OTHER`

- From version 2.6.23, Linux started using a scheduling algorithm for the `SCHED_OTHER` class known as **completely fair scheduler (CFS)**
- Schedule is based on a virtual run time value (`vruntime`), which records how long a thread has run so far
 - the virtual run time value is related to the physical run time and the priority of the thread
 - higher priorities **shrinks** the physical run time
- The scheduler selects for execution the thread with the smallest virtual run time value
 - a higher-priority thread that becomes ready to run can preempt a lower-priority thread
 - thus, I/O-bound threads eventually can preempt CPU-bound ones
- The Linux CFS scheduler provides an efficient algorithm for selecting which thread to run next, based on a red-black tree;

Bibliography

- Operating Systems: Internals and Design Principles, W. Stallings, Prentice-Hall International Editions, 7th Ed, 2012
 - Chapter 9: Uniprocessor Scheduling (sections 9.1 to 9.3)
 - Chapter 10: Multiprocessor and Real-Time Scheduling (section 10.3)
- Operating Systems Concepts, A. Silberschatz, P. Galvin and G. Gagne, John Wiley & Sons, 9th Ed, 2013
 - Chapter 6: Process scheduling (sections 6.1 to 6.3 and 6.7.1)
- Modern Operating Systems, A. Tanenbaum and H. Bos, Pearson Education Limited, 4th Ed, 2015
 - Chapter 2: Processes and Threads (section 2.4)



Sistemas de Operação / Fundamentos de Sistemas Operativos

Interprocess communication

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

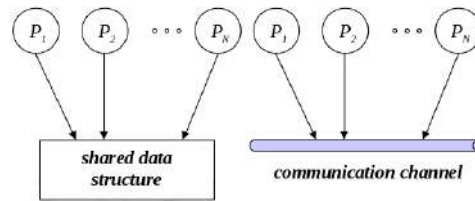
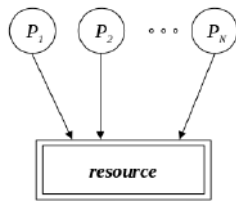
Outline

- ① Concepts
- ② Access primitives
- ③ Software solutions
- ④ Hardware solutions
- ⑤ Semaphores
- ⑥ Monitors
- ⑦ Message passing
- ⑧ Unix IPC primitives
- ⑨ Bibliography

Concepts

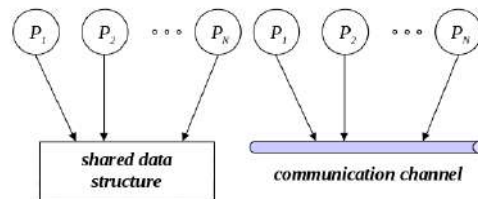
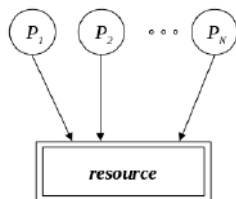
Independent and collaborative processes

- In a multiprogrammed environment, two or more processes can be:
 - **independent** – if they, from their creation to their termination, never explicitly interact
 - actually, there is an implicit interaction, as they compete for system resources
 - ex: jobs in a batch system; processes from different users
 - **cooperative** – if they share information or explicitly communicate
 - the **sharing** requires a **common address space**
 - **communication** can be done through a common address space or a **communication channel** connecting them



Concepts

Independent and collaborative processes (2)



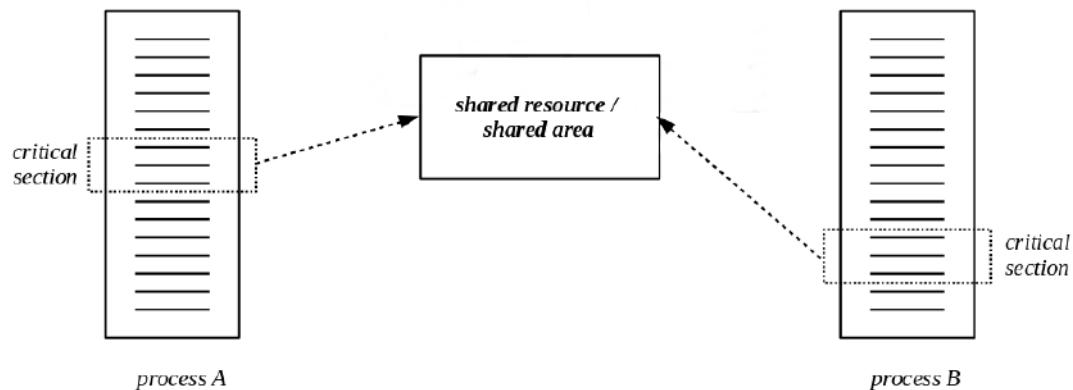
- **Independent processes** competing for a resource
- It is the **responsibility of the OS** to ensure the assignment of resources to processes is done in a controlled way, such that no information loss occurs
- In general, this imposes that only one process can use the resource at a time – **mutual exclusive access**

- **Cooperative processes** sharing information or communicating
- It is the **responsibility of the processes** to ensure that access to the shared area is done in a controlled way, such that no information loss occurs
- In general, this imposes that only one process can access the shared area at a time – **mutual exclusive access**
- The communication channel is typically a system resource, so processes compete for it

Concepts

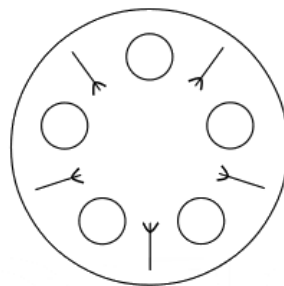
Critical section

- Having access to a resource or to a shared area actually means **executing the code** that does the access
- This section of code, if not properly protected, can result in **race conditions**
 - which can result in lost of information
 - It is called **critical section**
- Critical sections should execute in **mutual exclusion**



Philosopher dinner

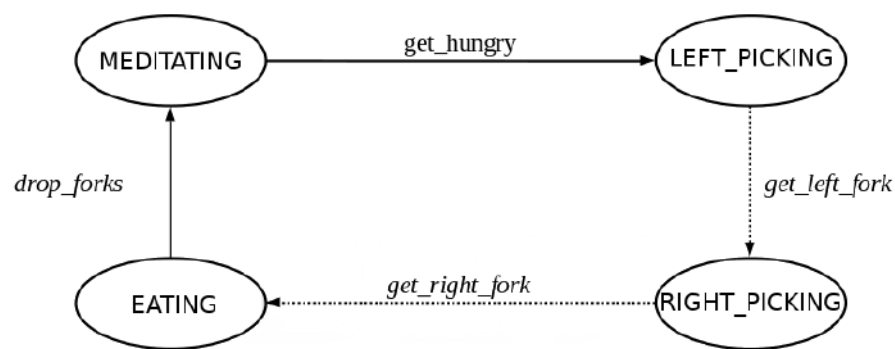
Problem statement



- 5 philosophers are seated around a table, with food in front of them
 - To eat, every philosopher needs two forks, the ones at her/his left and right sides
 - Every philosopher alternates periods in which she/he meditates with periods in which she/he eats
- Modeling every philosopher as a **different process or thread** and the forks as resources, **design a solution for the problem**

Philosopher dinner

A solution – state diagram



- This is a possible solution for the dining-philosopher problem
 - when a philosopher gets hungry, he/she first gets the left fork and then holds it while waits for the right one
- Let's look at an implementations of this solution!

Philosopher dinner

A solution – code

```
enum PHILO_STATE { MEDITATING, LEFT_PICKING, RIGHT_PICKING, EATING };

typedef struct TablePlace
{
    int state;
} TablePlace;

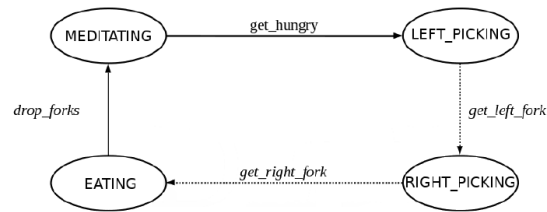
typedef struct Table
{
    int nplaces;
    TablePlace place[0];
} Table;

int set_table(unsigned int n);
int get_hungry(unsigned int f);
int get_left_fork(unsigned int f);
int get_right_fork(unsigned int f);
int drop_forks(unsigned int f);
```

-
- Let's execute the code

Philosopher dinner

A solution – a race condition



- This solution may work some times, but in general suffers from race conditions
- Let's look at a code snippet:
 - `get_right_fork:`

```
while (table->place[right(f)].state == EATING or  
       table->place[right(f)].state == RIGHT_PICKING);
```

Concepts

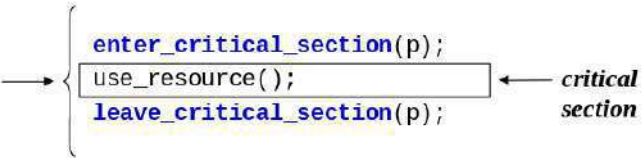
Deadlock and starvation

- Mutual exclusion in the access to a resource or shared area can result in
 - **deadlock** – when two or more processes are waiting forever to access to their respective critical section, waiting for events that can be demonstrated will never happen
 - operations are blocked
 - **starvation** – when one or more processes compete for access to a critical section and, due to a conjunction of circumstances in which new processes that exceed them continually arise, access is successively deferred
 - operations are continuously postponed

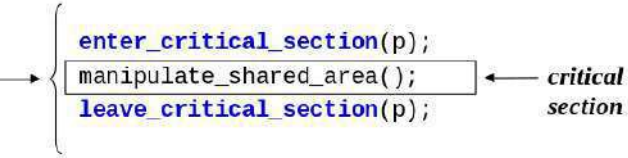
Access primitives

Access to a resource or to a shared area

```
/* processes competing for a resource -  $p = 0, 1, \dots, N-1$  */
void mainLoop (unsigned int p)
{
    forever
    {
        do_something();
        access_resource(p);
        do_something_else();
    }
}
```



```
/* shared data structure */
shared DATA d;
/* processes sharing data -  $p = 0, 1, \dots, N-1$  */
void mainLoop (unsigned int p)
{
    forever
    {
        do_something();
        access_shared_area(p);
        do_something_else();
    }
}
```




Access primitives

Producer-consumer example - producer

```
/* communicating data structure: FIFO of fixed size */
shared FIFO fifo;
/* producer processes -  $p = 0, 1, \dots, N-1$  */
void producer(unsigned int p)
{
    DATA val;
    bool done;


    forever
    {
        produce_data(&val);
        done = false;
        do
        {
            enter_critical_section(p);
            if (fifo.notFull())
            {
                fifo.insert(val);
                done = true;
            }
            leave_critical_section(p);
        } while (!done);
        do_something_else();
    }
}
```



Access primitives

Producer-consumer example - consumer

```
/* communicating data structure: FIFO of fixed size */
shared FIFO fifo;
/* consumer processes - p = 0, 1, ..., M-1 */
void consumer(unsigned int p)
{
    DATA val;
    bool done;
    forever
    {
        done = false;
        do
        {
            enter_critical_section(p);
            if (fifo.notEmpty())
            {
                fifo.retrieve(&val);
                done = true;
            }
            leave_critical_section(p);
        } while (!done);
        consume_data(val);
        do_something_else();
    }
}
```



Access primitives

Requirements

- Requirements that should be observed in accessing a critical section:
 - **Effective mutual exclusion** – access to the critical sections associated with the same resource, or shared area, can only be allowed to one process at a time, among all processes that compete for access
 - **Independence** on the number of intervening processes or on their relative speed of execution
 - a process **outside its critical section** cannot prevent another process from entering its own critical section
 - **No starvation** – a process requiring access to its critical section should not have to wait indefinitely
 - Length of stay inside a critical section should be necessarily **finite**

Access primitives

Types of solutions

- In general, a **memory location** is used to control access to the critical section
 - it works as a **binary flag**
- Two types of solutions: **software solutions** and **hardware solutions**
- **software solutions** – solutions that are based on the typical instructions used to access memory location
 - read and write are done by different instructions
 - interruption can occur between read and write
- **hardware solutions** – solutions that are based on special instructions to access the memory location
 - these instructions allow to read and then write a memory location in an atomic (uninterruptible) way

Software solutions

Constructing a solution - strict alternation

```
/* control data structure */
#define R    ...    /* process id = 0, 1, ..., R-1 */
shared unsigned int access_turn = 0;
void enter_critical_section(unsigned int own_pid)
{
    while (own_pid != access_turn);
}
void leave_critical_section(unsigned int own_pid)
{
    if (own_pid == access_turn)
        access_turn = (access_turn + 1) % R;
}
```

- Not a valid solution
 - Dependence on the relative speed of execution of the intervening processes
 - The process with less accesses imposes its rhythm to the others
 - A process outside the critical section can prevent another from entering there
 - If it is not its turn, a process has to wait, until its predecessor enters and give it access on leaving

Software solutions

Constructing a solution - 1st step

```
/* control data structure */
#define R 2 /* process id = 0, 1 */
shared bool is_in[R] = {false, false};
void enter_critical_section(unsigned int own_pid)
{
    unsigned int other_pid = 1 - own_pid;
    while (is_in[other_pid]);
    is_in[own_pid] = true;
}
void leave_critical_section(unsigned int own_pid)
{
    is_in[own_pid] = false;
}
```

- Not a valid solution
 - Mutual exclusion is not guaranteed

Software solutions

Constructing a solution - 1st step

```
/* control data structure */
#define R 2 /* process id = 0, 1 */
shared bool is_in[R] = {false, false};
void enter_critical_section(unsigned int own_pid)
{
    unsigned int other_pid = 1 - own_pid;
    while (is_in[other_pid]);
    is_in[own_pid] = true;
}
void leave_critical_section(unsigned int own_pid)
{
    is_in[own_pid] = false;
}
```

- Assume the following sequence of execution:
 - P_0 enters `enter_critical_section` and tests `is_in[1]` as being false
 - P_1 enters `enter_critical_section` and tests `is_in[0]` as being false
 - P_1 changes `is_in[1]` to true and enters its critical section
 - P_0 changes `is_in[0]` to true and enters its critical section
- Thus, both processes enter their critical sections
- It seems that the failure is a result of testing first the other's control variable and then change its own variable

Software solutions

Constructing a solution - 2nd step

```
/* control data structure */
#define R 2 /* process pid = 0, 1 */
shared bool want_enter[R] = {false, false};
void enter_critical_section (unsigned int own_pid)
{
    unsigned int other_pid = 1 - own_pid;
    want_enter[own_pid] = true;
    while (want_enter[other_pid]);
}
void leave_critical_section (unsigned int own_pid)
{
    want_enter[own_pid] = false;
}
```

- Not a valid solution
 - Mutual exclusion is guaranteed, but deadlock can occur

Software solutions

Constructing a solution - 2nd step

```
/* control data structure */
#define R 2 /* process pid = 0, 1 */
shared bool want_enter[R] = {false, false};
void enter_critical_section (unsigned int own_pid)
{
    unsigned int other_pid = 1 - own_pid;
    want_enter[own_pid] = true;
    while (want_enter[other_pid]);
}
void leave_critical_section (unsigned int own_pid)
{
    want_enter[own_pid] = false;
}
```

- Assume that:
 - P_0 enters `enter_critical_section` and sets `want_enter[0]` to true
 - P_1 enters `enter_critical_section` and sets `want_enter[1]` to true
 - P_1 tests `want_enter[0]` and, because it is true, keeps waiting to enter its critical section
 - P_0 tests `want_enter[1]` and, because it is true, keeps waiting to enter its critical section
- Thus, both processes enter deadlock
- To solve the deadlock at least one of the processes have to go back

Software solutions

Constructing a solution - 3rd step

```
/* control data structure */
#define R 2 /* process id = 0, 1 */
shared bool want_enter[R] = {false, false};
void enter_critical_section(unsigned int own_pid)
{
    unsigned int other_pid = 1 - own_pid;
    want_enter[own_pid] = true;
    while (want_enter[other_pid])
    {
        want_enter[own_pid] = false;
        random_delay();
        want_enter[own_pid] = true;
    }
}
void leave_critical_section(unsigned int own_pid)
{
    want_enter[own_pid] = false;
}
```

- An almost valid solution
 - The Ethernet protocol uses a similar approach to control access to the communication medium

Software solutions

Constructing a solution - 3rd step

```
/* control data structure */
#define R 2 /* process id = 0, 1 */
shared bool want_enter[R] = {false, false};
void enter_critical_section(unsigned int own_pid)
{
    unsigned int other_pid = 1 - own_pid;
    want_enter[own_pid] = true;
    while (want_enter[other_pid])
    {
        want_enter[own_pid] = false;
        random_delay();
        want_enter[own_pid] = true;
    }
}
void leave_critical_section(unsigned int own_pid)
{
    want_enter[own_pid] = false;
}
```

- An almost valid solution
 - The Ethernet protocol uses a similar approach to control access to the communication medium
- But, still not completely valid
 - Even if unlikely, deadlock and starvation can still be present
- The solution needs to be deterministic, not random

Software solutions

Dekker algorithm (1965)

```
#define R 2 /* process id = 0, 1 */
shared bool want_enter[R] = {false, false};
shared uint p_w_priority = 0;
void enter_critical_section(uint own_pid)
{
    uint other_pid = 1 - own_pid;
    want_enter[own_pid] = true;
    while (want_enter[other_pid])
    {
        if (own_pid != p_w_priority)
        {
            want_enter[own_pid] = false;
            while (own_pid != p_w_priority);
            want_enter[own_pid] = true;
        }
    }
}
void leave_critical_section(uint own_pid)
{
    uint other_pid = 1 - own_pid;
    p_w_priority = other_pid;
    want_enter[own_pid] = false;
}
```

Software solutions

Dekker algorithm (1965)

```
#define R 2 /* process id = 0, 1 */
shared bool want_enter[R] = {false, false};
shared uint p_w_priority = 0;
void enter_critical_section(uint own_pid)
{
    uint other_pid = 1 - own_pid;
    want_enter[own_pid] = true;
    while (want_enter[other_pid])
    {
        if (own_pid != p_w_priority)
        {
            want_enter[own_pid] = false;
            while (own_pid != p_w_priority);
            want_enter[own_pid] = true;
        }
    }
}
void leave_critical_section(uint own_pid)
{
    uint other_pid = 1 - own_pid;
    p_w_priority = other_pid;
    want_enter[own_pid] = false;
}
```

- The algorithm uses an alternation mechanism (on the priority) to solve the conflict
- Mutual exclusion in the access to the critical section is guaranteed
- Deadlock and starvation are not present
- No assumptions are done in the relative speed of the intervening processes
- However, it can **not be generalized** to more than 2 processes, satisfying all the requirements

Software solutions

Dijkstra algorithm (1966)

```
#define R    ...    /* process id = 0, 1, ..., R-1 */
shared uint want_enter[R] = {NO, NO, ... , NO};
shared uint p_w_priority = 0;
void enter_critical_section(uint own_pid)
{
    uint n;
    do
    {
        want_enter[own_pid] = WANT;
        while (own_pid != p_w_priority)
            if (want_enter[p_w_priority] == NO)
                p_w_priority = own_pid;
        want_enter[own_pid] = DECIDED;
        for (n = 0; n < R; n++)
            if (n != own_pid && want_enter[n] == DECIDED)
                break;
    } while (n < R);
}
void leave_critical_section(uint own_pid)
{
    p_w_priority = (own_pid + 1) % R;
    want_enter[own_pid] = NO;
}
```

- Works, but can suffer from [starvation](#)

Software solutions

Peterson algorithm (1981)

```
#define R    2    /* process id = 0, 1 */
shared bool want_enter[R] = {false, false};
shared uint last;
void enter_critical_section(uint own_pid)
{
    uint other_pid = 1 - own_pid;
    want_enter[own_pid] = true;
    last = other_pid;
    while ((want_enter[other_pid]) && (last == other_pid));
}
void leave_critical_section(uint own_pid)
{
    want_enter[own_pid] = false;
}
```

- The Peterson algorithm uses the order of arrival to solve conflicts
 - Each process has to write the other's ID in a shared variable (last)
 - The subsequent reading allows to determine which was the last one

Software solutions

Peterson algorithm (1981)

```
#define R 2 /* process id = 0, 1 */
shared bool want_enter[R] = {false, false};
shared uint last;
void enter_critical_section(uint own_pid)
{
    uint other_pid = 1 - own_pid;
    want_enter[own_pid] = true;
    last = other_pid;
    while ((want_enter[other_pid]) && (last == other_pid));
}
void leave_critical_section(uint own_pid)
{
    want_enter[own_pid] = false;
}
```

- The Peterson algorithm uses the order of arrival to solve conflicts
 - Each process has to write the other's ID in a shared variable (last)
 - The subsequent reading allows to determine which was the last one
- It is a valid solution
 - Guarantees mutual exclusion
 - Avoids deadlock and starvation
 - Makes no assumption about the relative speed of intervening processes

Software solutions

Generalized Peterson algorithm (1981)

```
#define R ... /* process id = 0, 1, ..., R-1 */
shared int level[R] = {-1, -1, ..., -1};
shared int last[R-1];
void enter_critical_section(uint own_pid)
{
    for (uint i = 0; i < R-1; i++)
    {
        level[own_pid] = i;
        last[i] = own_pid;
        do
        {
            test = false;
            for (uint j = 0; j < R; j++)
                if (j != own_pid)
                    test = test || (level[j] >= i);
        } while (test && (last[i] == own_pid));
    }
}
void leave_critical_section(int own_pid)
{
    level[own_pid] = -1;
}
```

- Can be generalized to more than two processes
 - The general solution is similar to a waiting queue

Hardware solutions

disabling interrupts

- *Uniprocessor computational system*
 - The switching of processes, in a multiprogrammed environment, is always caused by an external device:
 - **real time clock (RTC)** – cause the time-out transition in preemptive systems
 - **device controller** – can cause the preempt transitions in case of waking up of a higher priority process
 - In any case, interruptions of the processor
 - Thus, access in mutual exclusion can be implemented disabling interrupts
 - Only valid in kernel
 - Malicious or buggy code can completely block the system
- *Multiprocessor computational system*
 - Disabling interrupts in one processor has no effect

Hardware solutions

special instructions – TAS

```
shared bool flag = false;

bool test_and_set(bool * flag)
{
    bool prev = *flag;
    *flag = true;
    return prev;
}

void lock(bool * flag)
{
    while (test_and_set(flag);
}

void unlock(bool * flag)
{
    *flag = false;
}
```

- The **test_and_set** function, if implemented **atomically** (without interruptions), can be used to construct the **lock** (**enter critical section**) primitive
- In the instruction set of some of the current processors, there is an atomic instruction implementing this behavior
- Surprisingly, it is often called **TAS** (**test and set**)

Hardware solutions

special instructions – CAS

```
shared int value = 0;

int compare_and_swap(int * value,
    int expected, int new_value)
{
    int v = *value;
    if (*value == expected)
        *value = new_value;
    return v;
}

void lock(int * flag)
{
    while (compare_and_swap(&flag,
        0, 1) != 0);
}

void unlock(bool * flag)
{
    *flag = 0;
}
```

- The `compare_and_swap` function, if implemented **atomically** (without interruptions), can be used to construct the **lock** (enter critical section) primitive
- In the instruction set of some of the current processors, there is an atomic instruction implementing that behavior
- In some instruction sets, there is a `compare_and_set` variant this returns a bool

Hardware solutions

Busy waiting

- The previous solutions suffer from **busy waiting**
 - The **lock** primitive is in the active state (using the CPU) while waiting
 - It is often referred to as a **spinlock**, as the process spins around the variable while waiting for access
- In **uniprocessor systems**, busy waiting is unwanted, as there is
 - **loss of efficiency** – the time quantum of a process is used for nothing
 - **risk of deadlock** – if a higher priority process calls lock while a lower priority process is inside its critical section, none of them can proceed
- In **multiprocessor systems** with shared memory, busy waiting can be less critical
 - switching processes cost time, that can be higher than the time spent by the other process inside its critical section

Hardware solutions

Block and wake up

- In general, at least in uniprocessor systems, there is the requirement of blocking a process while it is waiting for entering its critical section

```
#define R    ...    /* process id = 0, 1, ..., R-1 */
shared unsigned int access = 1;
void enter_critical_section(unsigned int own_pid)
{
    if (access == 0) block(own_pid);
    else access -= 1;
}
void leave_critical_section(unsigned int own_pid)
{
    if (there_are_blocked_processes) wake_up_one();
    else access += 1;
}
```

→ { *atomic operation*
(can not be interrupted)

→ { *atomic operation*
(can not be interrupted)

- Atomic operations are still required

Semaphores

Definition

- A **semaphore** is a synchronization mechanism, defined by a data type plus two atomic operations, **down** and **up**
- Data type:

```
typedef struct
{
    unsigned int val;    /* can not be negative */
    PROCESS *queue;     /* queue of waiting blocked processes */
} SEMAPHORE;
```

- Operations:
 - down**
 - block process if `val` is zero
 - decrement `val` otherwise
 - up**
 - if `queue` is not empty, wake up one waiting process (accordingly to a given policy)
 - increment `val` otherwise
- Note that `val` can only be manipulated through these operations
 - It is not possible to check the value of `val`

Semaphores

An implementation of semaphores

```
/* array of semaphores defined in kernel */
#define R ... /* semid = 0, 1, ..., R-1 */
static SEMAPHORE sem[R];
void sem_down(unsigned int semid)
{
    disable_interruptions;
    if (sem[semid].val == 0)
        block_on_sem(getpid(), semid);
    sem[semid].val -= 1;
    enable_interruptions;
}
void sem_up(unsigned int semid)
{
    disable_interruptions;
    sem[semid].val += 1;
    if (sem[semid].queue != NULL)
        wake_up_one_on_sem(semid);
    enable_interruptions;
}
```

- Internally, the `block_on_sem` function must enable interruptions
- This implementation is typical of uniprocessor systems. Why?

- Semaphores can be binary or not binary
- How to implement **mutual exclusion** using semaphores?
 - Using a **binary** semaphore

Semaphores

Bounded-buffer problem – solving using semaphores

```
shared FIFO fifo; /* fixed-size FIFO memory */
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        bool done = false;
        do
        {
            if (fifo.notFull())
            {
                fifo.insert(data);
                done = true;
            }
        } while (!done);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        bool done = false;
        do
        {
            if (fifo.notEmpty())
            {
                fifo.retrieve(&data);
                done = true;
            }
        } while (!done);
        consume_data(data);
        do_something_else();
    }
}
```

- This solution can suffer **race conditions**

Semaphores

Bounded-buffer problem – solving using semaphores

```
shared FIFO fifo; /* fixed-size FIFO memory */
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        bool done = false;
        do
        {
            if (fifo.notFull())
            {
                lock(p);
                fifo.insert(data);
                done = true;
                unlock(p);
            }
        } while (!done);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        bool done = false;
        do
        {
            if (fifo.notEmpty())
            {
                lock(c);
                fifo.retrieve(&data);
                done = true;
                unlock(c);
            }
        } while (!done);
        consume_data(data);
        do_something_else();
    }
}
```

- Mutual exclusion is guaranteed, but suffers from busy waiting

Semaphores

Bounded-buffer problem – solving using semaphores

```
shared FIFO fifo; /* fixed-size FIFO memory */
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        bool done = false;
        do
        {
            if (fifo.notFull())
            {
                fifo.insert(data);
                done = true;
            }
        } while (!done);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        bool done = false;
        do
        {
            if (fifo.notEmpty())
            {
                fifo.retrieve(&data);
                done = true;
            }
        } while (!done);
        consume_data(data);
        do_something_else();
    }
}
```

- How to implement using semaphores?
 - guaranteeing mutual exclusion and absence of busy waiting

Semaphores

Bounded-buffer problem – solving using semaphores

```
shared FIFO fifo;      /* fixed-size FIFO memory */
shared sem access;     /* semaphore to control mutual exclusion */
shared sem nslots;     /* semaphore to control number of available slots */
shared sem nitems;     /* semaphore to control number of available items */

/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA val;

    forever
    {
        produce_data(&val);
        sem_down(nslots);
        sem_down(access);
        fifo.insert(val);
        sem_up(access);
        sem_up(nitems);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA val;

    forever
    {
        sem_down(nitems);
        sem_down(access);
        fifo.retrieve(&val);
        sem_up(access);
        sem_up(nslots);
        consume_data(val);
        do_something_else();
    }
}
```

- `fifo.notEmpty()` and `fifo.notFull()` are no longer necessary. Why?
- What are the initial values of the semaphores?

Semaphores

Bounded-buffer problem – wrong solution

```
shared FIFO fifo;      /* fixed-size FIFO memory */
shared sem access;     /* semaphore to control mutual exclusion */
shared sem nslots;     /* semaphore to control number of available slots */
shared sem nitems;     /* semaphore to control number of available items */

/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA val;

    forever
    {
        produce_data(&val);
        sem_down(access);
        sem_down(nslots);
        fifo.insert(val);
        sem_up(access);
        sem_up(nitems);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA val;

    forever
    {
        sem_down(nitems);
        sem_down(access);
        fifo.retrieve(&val);
        sem_up(access);
        sem_up(nslots);
        consume_data(val);
        do_something_else();
    }
}
```

- One can easily make a mistake
- What is wrong with this solution? It can cause **deadlock**

Semaphores

Analysis of semaphores

- Concurrent solutions based on semaphores have advantages and disadvantages
- **Advantages:**
 - **support at the operating system level**– operations on semaphores are implemented by the kernel and made available to programmers as system calls
 - **general**– they are low level constructions and so they are versatile, being able to be used in any type of solution
- **Disadvantages:**
 - **specialized knowledge**– the programmer must be aware of concurrent programming principles, as race conditions or deadlock can be easily introduced
 - See the previous example, as an illustration of this

Monitors

Introduction

- A problem with semaphores is that they are used both to implement **mutual exclusion** and for **synchronization** between processes
 - Being low level primitives, they are applied in a **bottom-up** perspective
 - if required conditions are not satisfied, processes are blocked before they enter their critical sections
 - this approach is prone to errors, mainly in complex situations, as synchronization points can be scattered throughout the program
 - A higher level approach should followed a **top-down** perspective
 - processes must first enter their critical sections and then block if continuation conditions are not satisfied
 - A solution is to introduce a (concurrent) construction at the programming language level that deals with mutual exclusion and synchronization separately
-
- A **monitor** is a synchronization mechanism, independently proposed by Hoare and Brinch Hansen, supported by a (concurrent) programming language
 - It is composed of an internal data structure, initialization code and a number of accessing primitives

Monitors

Definition

```
monitor example
{
    /* internal shared data structure */
    DATA data;

    condition c; /* condition variable */

    /* access methods */
    method_1 (...)
    {
        ...
    }

    method_2 (...)
    {
        ...
    }

    ...

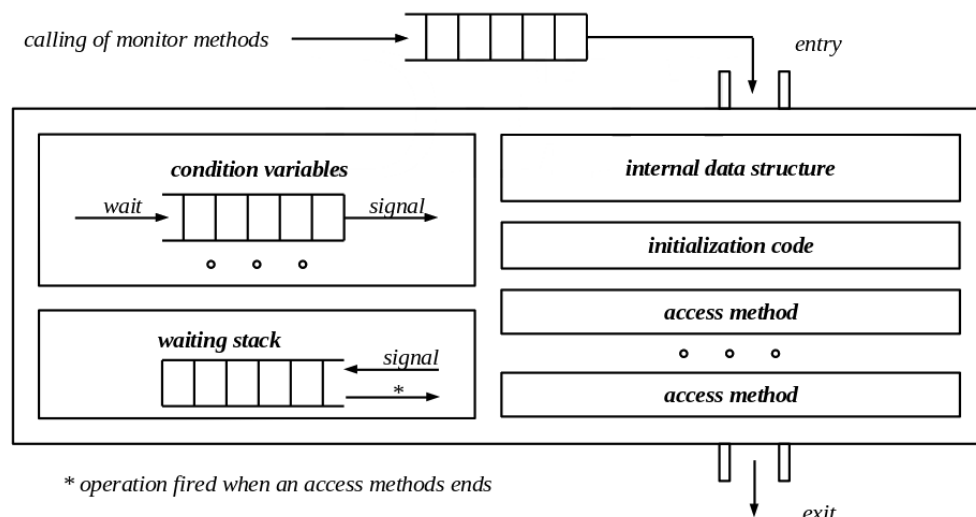
    /* initialization code */
    ...
}
```

- An application is seen as a set of threads that compete to access the **shared data** structure
- This shared data can only be accessed through the access methods
- Every method is executed in **mutual exclusion**
- If a thread calls an access method while another thread is inside another access method, its execution is blocked until the other leaves
- Synchronization between threads is possible through **condition variables**
- Two operation on them are possible:
 - **wait** – the thread is blocked and put outside the monitor
 - **signal** – if there are threads blocked, one is waked up. *Which one?*

Monitors

Hoare monitor

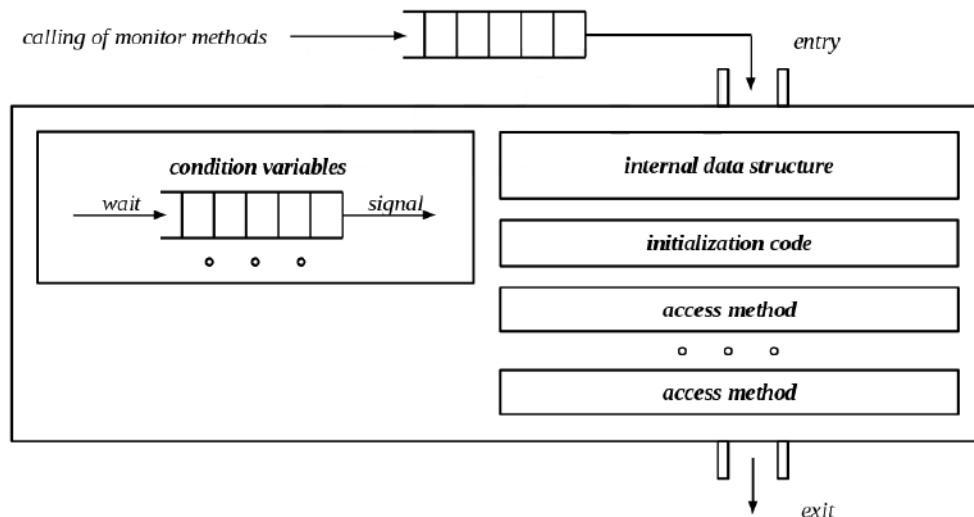
- What to do when **signal** occurs?
- **Hoare monitor** – the thread calling signal is put out of the monitor, so the just waked up thread can proceed
 - quite general, but its implementation requires a stack where the blocked thread is put



Monitors

Brinch Hansen monitor

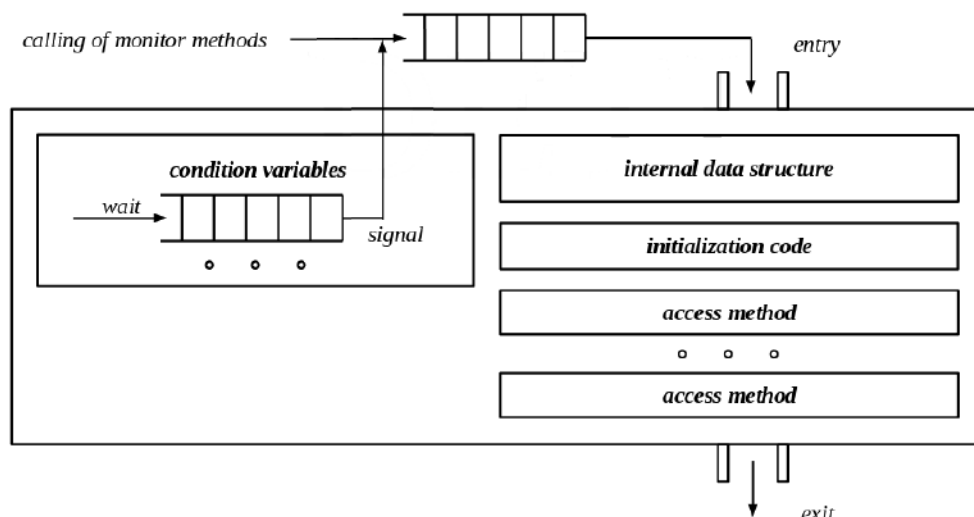
- What to do when **signal** occurs?
- **Brinch Hansen monitor** – the thread calling signal immediately leaves the monitor (signal is the last instruction of the monitor method)
 - easy to implement, but quite restrictive (only one signal allowed in a method)



Monitors

Lampson / Redell monitor

- What to do when **signal** occurs?
- **Lampson / Redell monitor** – the thread calling signal continues its execution and the just waked up thread is kept outside the monitor, competing for access
 - easy to implement, but can cause starvation



Monitors

Bounded-buffer problem – solving using monitors

```
shared FIFO fifo; /* fixed-size FIFO memory */
shared mutex access; /* mutex to control mutual exclusion */
shared cond nslots; /* condition variable to control availability of slots */
shared cond nitems; /* condition variable to control availability of items */

/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        lock(access);
        if/while (fifo.isFull())
        {
            wait(nslots, access);
        }
        fifo.insert(data);
        signal(nitems);
        unlock(access);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        lock(access);
        if/while (fifo.isEmpty())
        {
            wait(nitems, access);
        }
        fifo.retrieve(&data);
        signal(nslots);
        unlock(access);
        consume_data(data);
        do_something_else();
    }
}
```

- What is the initial state of the mutex?

Message-passing

Introduction

- Processes can communicate exchanging messages
 - A general communication mechanism, not requiring explicit shared memory, that includes both communication and synchronization
 - Valid for uniprocessor and multiprocessor systems
- Two operations are required:
 - **send** and **receive**
- A communication link is required
 - That can be categorized in different ways:
 - Direct or indirect communication
 - Synchronous or asynchronous communication
 - Type of buffering

Message-passing

Direct and indirect communication

- **Symmetric direct communication**

- A process that wants to communicate must explicitly name the receiver or sender
 - `send(P, msg)` – send message `msg` to process `P`
 - `receive(P, msg)` – receive message `msg` from process `P`
- A communication link in this scheme has the following properties:
 - it is established automatically between a pair of communicating processes
 - it is associated with exactly two processes
 - between a pair of communicating processes there exist exactly one link

- **Asymmetric direct communication**

- Only the sender must explicitly name the receiver
 - `send(P, msg)` – send message `msg` to process `P`
 - `receive(id, msg)` – receive message `msg` from any process

Message-passing

Direct and indirect communication

- **Indirect communication**

- The messages are sent to and received from mailboxes, or ports
 - `send(M, msg)` – send message `msg` to mailbox `M`
 - `receive(M, msg)` – receive message `msg` from mailbox `M`
- A communication link in this scheme has the following properties:
 - it is only established if the pair of communicating processes has a shared mailbox
 - it may be associated with more than two processes
 - between a pair of processes there may exist more than one link (a mailbox per each)
- The problem of two or more processes trying to receive a message from the same mailbox
 - Is it allowed?
 - If allowed, which one will succeed?

Message-passing

Synchronization

- From a synchronization point of view, there are different design options for implementing **send** and **receive**
 - **Blocking send**– the sending process blocks until the message is received by the receiving process or by the mailbox
 - **Nonblocking send**– the sending process sends the message and resumes operation
 - **Blocking receive**– the receiver blocks until a message is available
 - **Nonblocking receive**– the receiver retrieves either a valid message or the indication that no one exists
- Different combinations of send and receive are possible

Message-passing

Buffering

- There are different design options for implementing the link supporting the communication
 - **Zero capacity** – there is no queue
 - the sender must block until the recipient receives the message
 - **Bounded capacity** – the queue has finite length
 - if the queue is full, the sender must block until space is available
 - **Unbounded capacity** – the queue has (potentially) infinite length

Message-passing

Bounded-buffer problem – solving using messages

```
shared MailBox mbox;
```

```
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    MESSAGE msg;

    forever
    {
        produce_data(&data);
        make_message(msg, data);
        send(msg, mbox);
        do_something_else();
    }
}
```

```
/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    MESSAGE msg;

    forever
    {
        receive(msg, mbox);
        extract_data(data, msg);
        consume_data(data);
        do_something_else();
    }
}
```

- There is no need to deal with mutual exclusion and synchronization explicitly
 - the `send` and `receive` primitives take care of it

Unix IPC primitives

POSIX support for monitor implementation

- Standard POSIX, IEEE 1003.1c, defines a programming interface (API) for the creation and synchronization of threads
 - In unix, this interface is implemented by the `pthread` library
- It allows for the implementation of monitors in C/C++
 - Using mutexes and condition variables
 - Note that they are of the [Lampson / Redell](#) type
- Some of the available functions:
 - `pthread_create` – creates a new thread; similar to `fork`
 - `pthread_exit` – equivalent to `exit`
 - `pthread_join` – equivalent a `waitpid`
 - `pthread_self` – equivalent a `getpid()`
 - `pthread_mutex_*` – manipulation of mutexes
 - `pthread_cond_*` – manipulation of condition variables
 - `pthread_once` – initialization

Unix IPC primitives

Semaphores

- **System V semaphores**
 - creation: `semget`
 - down and up: `semop`
 - other operations: `semctl`
- **POSIX semaphores**
 - down and up
 - `sem_wait`, `sem_trywait`, `sem_timedwait`, `sem_post`
 - Two types: named and unnamed semaphores
 - Named semaphores
 - `sem_open`, `sem_close`, `sem_unlink`
 - created in a virtual filesystem (e.g., `/dev/sem`)
 - unnamed semaphores – memory based
 - `sem_init`, `sem_destroy`
 - execute `man sem_overview` for an overview

Unix IPC primitives

Message-passing

- **System V implementation**
 - Defines a message queue where messages of different types (a positive integer) can be stored
 - The send operation blocks if space is not available
 - The receive operation has an argument to specify the type of message to receive: a given type, any type or a range of types
 - The oldest message of given type(s) is retrieved
 - Can be blocking or nonblocking
 - see system calls: `msgget`, `msgsnd`, `msgrcv`, and `msgctl`
- **POSIX message queue**
 - Defines a priority queue
 - The send operation blocks if space is not available
 - The receive operation removes the oldest message with the highest priority
 - Can be blocking or nonblocking
 - see functions: `mq_open`, `mq_send`, `mq_receive`, ...

Unix IPC primitives

Shared memory

- Address spaces of processes are independent
- But address spaces are virtual
- The same physical region can be mapped into two or more virtual regions
- This is managed as a resource by the operating system
- **System V shared memory**
 - creation – `shmget`
 - mapping and unmapping – `shmat`, `shmdt`
 - other operations – `shmctl`
- **POSIX shared memory**
 - creation - `shm_open`, `ftruncate`
 - mapping and unmapping - `mmap`, `munmap`
 - other operations - `close`, `shm_unlink`, `fchmod`, ...

Bibliography

- Operating Systems: Internals and Design Principles, W. Stallings, Prentice-Hall International Editions, 7th Ed, 2012
 - Chapter 5: Concurrency: mutual exclusion and synchronization (sections 5.1 to 5.5)
- Operating Systems Concepts, A. Silberschatz, P. Galvin and G. Gagne, John Wiley & Sons, 9th Ed, 2013
 - Chapter 3: Processes (section 3.4)
 - Chapter 4: Process synchronization (sections 5.1 to 5.8)
- Modern Operating Systems, A. Tanenbaum and H. Bos, Pearson Education Limited, 4th Ed, 2015
 - Chapter 2: Processes and Threads (section 2.3)



Sistemas de Operação / Fundamentos de Sistemas Operativos

Deadlock

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

Outline

- ① Introduction
- ② Deadlock prevention
- ③ Deadlock avoidance
- ④ Deadlock detection
- ⑤ Bibliography

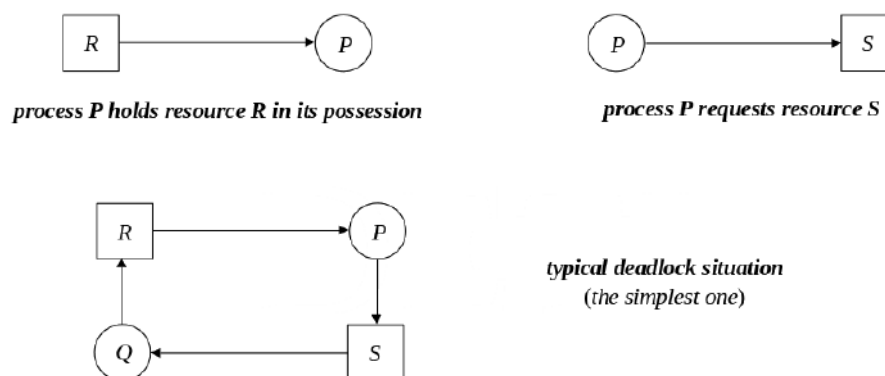
Deadlock

Introduction

- Generically, a **resource** is something a process needs in order to proceed with its execution
 - **physical components of the computational system** (processor, memory, I/O devices, etc.)
 - **common data structures** defined at the operating system level (PCT, communication channels, etc,) or among processes of a given application
- Resources can be:
 - **preemptable** – if they can be withdraw from the processes that hold them
 - ex: processor, memory regions used by a process address space
 - **non-preemptable** – if they can only be released by the processes that hold them
 - ex: a file, a shared memory region that requires exclusive access for its manipulation
- For this topic, only non-preemptable resources are relevant

Deadlock

Illustrating deadlock

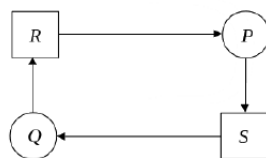


- P needs S to proceed, which is on possession of Q
- Q needs R to proceed, which is on possession of P
- What are the conditions for the occurrence of deadlock?

Deadlock

Necessary conditions for deadlock

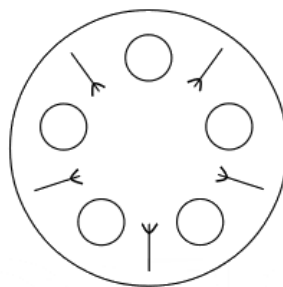
- It can be proved that when deadlock occurs 4 conditions are necessarily observed:
 - **mutual exclusion** – only one process may use a resource at a time
 - if another process requests it, it must wait until it is released
 - **hold and wait** – A process waits for some resources while holding others at the same time
 - **no preemption** – resources are non-preemptable
 - only the process holding a resource can release it, after completing its task
 - **circular wait** – a set of waiting processes must exist such that each one is waiting for resources held by other processes in the set
 - there are loops in the graph



*typical deadlock situation
(the simplest one)*

Deadlock

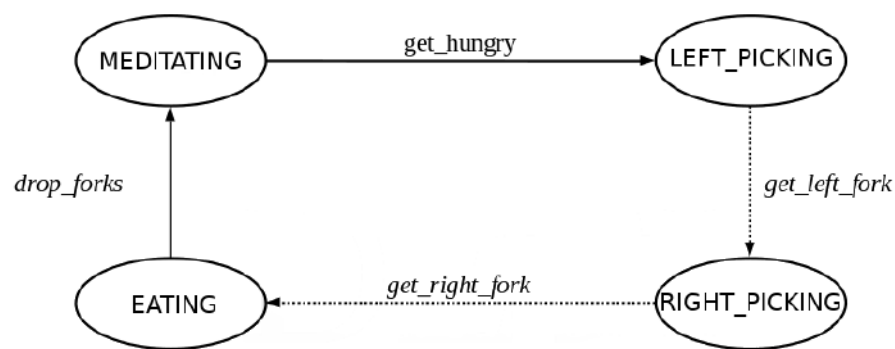
Illustrating with the philosopher dinner problem



- 5 philosophers are seated around a table, with food in front of them
 - To eat, every philosopher needs two forks, the ones at her/his left and right sides
 - Every philosopher alternates periods in which she/he meditates with periods in which she/he eats
- Modelling every philosopher as a **different process or thread** and the forks as resources, **design a solution for the problem**

Philosopher dinner

Solution 1 – state diagram



- This is a possible solution for the dining-philosopher problem
 - when a philosopher gets hungry, he/she first gets the left fork and then holds it while waits for the right one
- Let's look at an implementations of this solution!

Philosopher dinner

Solution 1 – code

```
enum PHILO_STATE {MEDITATING, HUNGRY, HOLDING, EATING};
enum FORK_STATE {DROPPED, TAKEN};

typedef struct TablePlace
{
    int philo_state;
    int fork_state;
    cond fork_available;
} TablePlace;

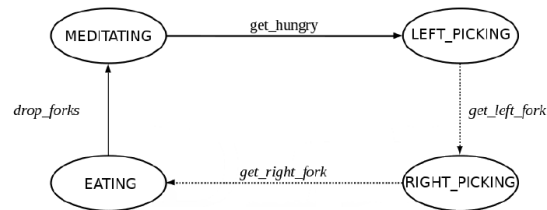
typedef struct Table
{
    mutex locker;
    int nplaces;
    TablePlace place[0];
} Table;

int set_table(unsigned int n, FILE *logp);
int get_hungry(unsigned int f);
int get_left_fork(unsigned int f);
int get_right_fork(unsigned int f);
int drop_forks(unsigned int f);
```

- Let's execute the code

Philosopher dinner

Solution 1 – deadlock conditions



- This solution works some times, but can suffer from deadlock
- Let's identify the four necessary conditions
 - **mutual exclusion** – the forks are not sharable at the same time
 - **hold and wait** – each philosopher, while waiting to acquire the right fork, holds the left one
 - **no preemption** – only the philosophers can release the fork(s) in their possession
 - **circular wait** – if all philosopher acquire the left fork, there is a chain in which every philosopher waits for a fork in possession of another philosopher

Deadlock prevention

Definition

- From the definition
deadlock \implies
mutual exclusion **and**
hold and wait **and**
no preemption **and**
circular wait
- Which is equivalent to
not mutual exclusion **or**
not hold and wait **or**
not no preemption **or**
not circular wait
 \implies **not** deadlock
- So, if in the solution of a concurrent problem at least one of the necessary conditions can never hold, there is no possibility of deadlock
- This is called **deadlock prevention**
 - the prevention lies on the application side

Deadlock prevention

Denying the necessary conditions

- Denying the **mutual exclusion** condition is only possible if resources are shareable at the same time
 - Otherwise race conditions can occur
 - Denying the **preemption** condition is only possible if resources are preemptable
 - Which is often not the case
 - Thus, in general, only the other conditions (**hold-and-wait** and **circular wait**) are used to implement deadlock prevention
-
- In the dining-philosopher problem, the forks are not shareable at the same time
 - In the dining-philosopher problem, a fork cannot be taken away from whoever has it

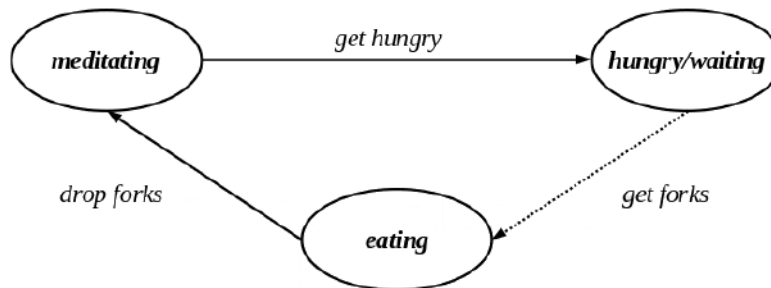
Deadlock prevention

Denying the necessary conditions (2)

- Denying the **hold-and-wait** condition can be done if a process requests all required resources at once
 - In this solution, starvation can occur
 - **Aging** mechanisms are often used to solve starvation
- In the dining-philosopher problem, the two forks must be acquired at once

Philosopher dinner

Solution 2 – state diagram



- This solution is equivalent to the one proposed by Dijkstra
- Every philosopher, when wants to eat, gets the two forks at the same time
- If they are not available, the philosopher waits in the hungry/waiting state
- Starvation is not avoided

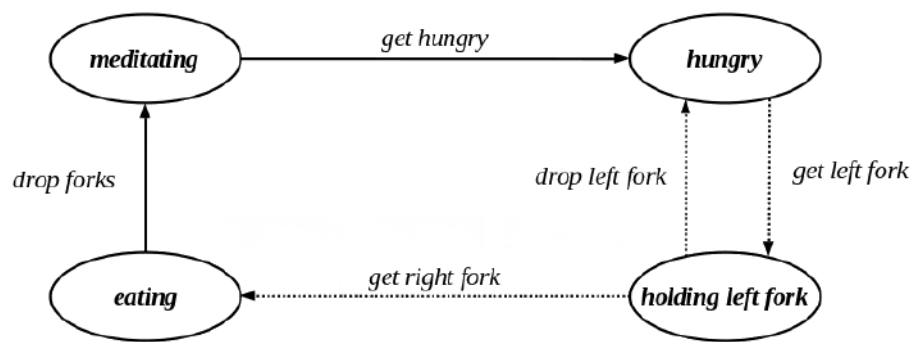
Deadlock prevention

Denying the necessary conditions (3)

- Denying the **hold and wait** condition can also be done if a process releases the already acquired resource(s) if it fails acquiring the next one
 - Later on it can try the acquisition again
- In the dining-philosopher problem, a philosopher must release the left fork if she/he fails acquiring the right one
 - In this solution, starvation and busy waiting can occur
 - Aging mechanisms are often used to solve starvation
 - To avoid busy waiting, the process should block and be waked up when the resource is released

Philosopher dinner

Solution 3 – state diagram



- When a philosopher gets hungry, she/he first acquire the left fork
- Then she/he tries to acquired the right one, releasing the left if she/he fails and returning to the hungry state
- **busy waiting** and **starvation** were not avoided in this solution

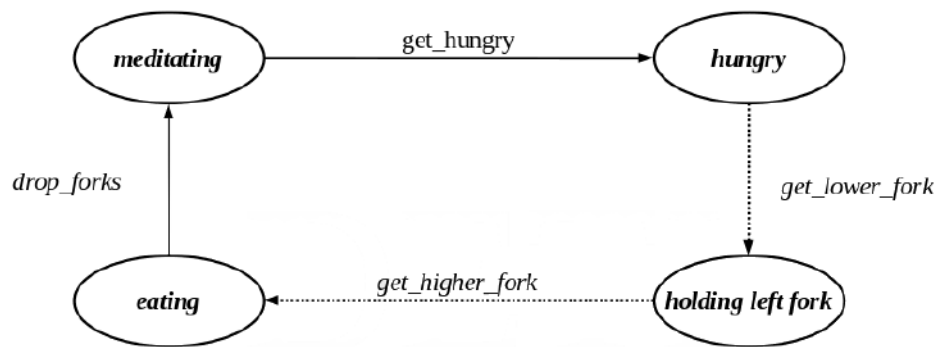
Deadlock prevention

Denying the necessary conditions (4)

- Denying the **circular wait** condition can be done assigning a different numeric id to every resource and imposing that the acquisition of resources have to be done either in ascending or descending order
 - This way the circular chain is always avoided
 - Starvation is not avoided
- In the dining-philosopher problem, this can be done imposing that one of the philosophers acquires first the right fork and then the left one
 - Show it!

Philosopher dinner

Solution 4 – state diagram



- Philosophers are numbered from 0 to $N - 1$
- Every fork is assigned an id, equal to the id of the philosopher at its left, for instance
- Every philosopher, acquires first the fork with the lower id
- This way, philosophers 0 to $N - 2$ acquire first the left fork, while philosopher $N - 1$ acquires first the right one

Deadlock avoidance

Definition

- **Deadlock avoidance** is less restrictive than deadlock prevention
 - None of the deadlock conditions is denied a priori
 - The resource system is monitored in order to decide what to do in terms of resource allocation
 - Requires **knowledge in advance** of maximum process resource requests
 - The intervening processes have to **declare at start** their needs in terms of resources
- Two possible approaches
 - **Process Initiation Denial**
 - Do not start a process if its demands might lead to deadlock
 - **Resource Allocation Denial**
 - Do not grant an incremental resource request to a process if this allocation might lead to deadlock

Deadlock avoidance

Process initiation denial

- The system prevents a new process to start if its termination can not be guaranteed
- Let
 - $R = (R_1, R_2, \dots, R_n)$ be a vector of the total amount of each resource
 - P be the set of processes competing for resources
 - C_p be a vector of the total amount of each resource declared by process $p \in P$
- A new process q ($q \notin P$) is only started if

$$C_q \leq R - \sum_{p \in P} C_p$$

- It is a quite restrictive approach

Deadlock avoidance

Resource allocation denial

- A new resource is allocated to a process if and only if there is at least one sequence of future allocations that does not result in deadlock
 - In such cases, the system is said to be in a **safe state**
- Let
 - $R = (R_1, R_2, \dots, R_n)$ be a vector of the total amount of each resource
 - $V = (V_1, V_2, \dots, V_n)$ be a vector of the amount of each resource available
 - P be the set of processes competing for resources
 - C_p be a vector of the total amount of each resource declared by process $p \in P$
 - A_p be a vector of the amount of each resource already allocated to process $p \in P$
- A new request of a process q is only granted if, after it, there is a sequence $s(k)$, with $s(k) \in P$ and $k = 1, 2, \dots, |P|$, of processes, such that

$$C_{s(k)} - A_{s(k)} = V + \sum_{m=1}^{k-1} A_{s(m)}$$

- This approach is called the **banker's algorithm**

Deadlock avoidance

Banker's algorithm

		R1	R2	R3	R4
total resources		6	5	7	6
available resources		3	1	1	2
resources declared	P1	3	3	2	2
	P2	1	2	3	4
	P3	1	3	5	0
resources allocated	P1	1	2	2	1
	P2	1	0	3	3
	P3	1	2	1	0
resources requestable	P1	2	1	0	1
	P2	0	2	0	1
	P3	0	1	4	0

- Consider the system state described by the table. Is it a safe state?
 - P2 may still request 2 R2, but only one is available
 - P3 may still request 4 R3, but only one is available
 - All resources that P1 can still request are available

Deadlock avoidance

Banker's algorithm (2)

		R1	R2	R3	R4
total resources		6	5	7	6
available resources		3	1	1	2
resources declared	P1	3	3	2	2
	P2	1	2	3	4
	P3	1	3	5	0
resources allocated	P1	1	2	2	1
	P2	1	0	3	3
	P3	1	2	1	0
resources requestable	P1	2	1	0	1
	P2	0	2	0	1
	P3	0	1	4	0
new request		—	—	—	—

- Consider the following sequence:
 - P1 requests all the resources it can still; the request is granted; then terminates
 - P2 requests all the resources it can still; the request is granted; then terminates
 - P3 requests all the resources it can still; the request is granted; then terminates

Deadlock avoidance

Banker's algorithm (3)

		R1	R2	R3	R4
total resources		6	5	7	6
available resources		3	1	1	2
resources declared	P1	3	3	2	2
	P2	1	2	3	4
	P3	1	3	5	0
resources allocated	P1	1	2	2	1
	P2	1	0	3	3
	P3	1	2	1	0
resources requestable	P1	2	1	0	1
	P2	0	2	0	1
	P3	0	1	4	0
new request	P3	0	0	2	0

- If P3 requests 2 resources of type R3, the grant is postponed. Why?
 - Because only 1 is available

Deadlock avoidance

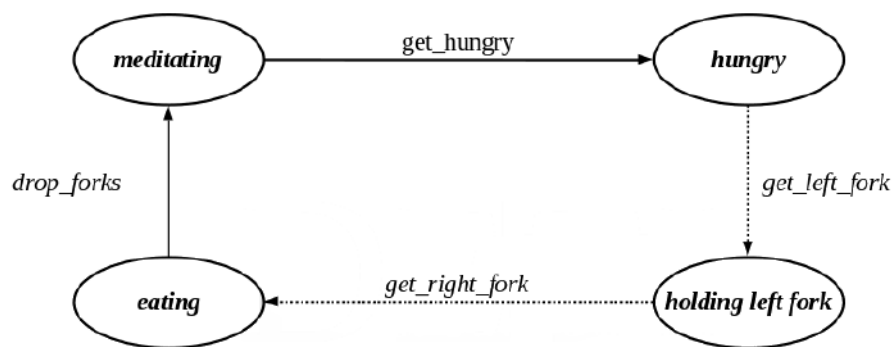
Banker's algorithm (4)

		R1	R2	R3	R4
total resources		6	5	7	6
available resources		3	1	1	2
resources declared	P1	3	3	2	2
	P2	1	2	3	4
	P3	1	3	5	0
resources allocated	P1	1	2	2	1
	P2	1	0	3	3
	P3	1	2	1	0
resources requestable	P1	2	1	0	1
	P2	0	2	0	1
	P3	0	1	4	0
new request	P3	0	1	0	0

- If P3 requests 1 resource of type R2, the grant is also postponed. Why?
 - Because, if the grant is given, the system transitions to an unsafe state. Show it.

Deadlock avoidance

Banker's algorithm - example



- Every philosopher first gets the left fork and then gets the right one
- However, in a specific situation the request of the left fork is postponed
 - What situation? Why?

Deadlock detection

Definition

- No deadlock-prevention or deadlock-avoidance is used
 - So, deadlock situations may occur
 - The state of the system should be examined to determine whether a deadlock has occurred
 - A recover from deadlock procedure should exist and be applied
-
- What to do?
 - In a quite naive approach, the problem can simply be ignored
 - Otherwise, the circular chain of processes and resources need to be broken

Deadlock detection

Recover procedure

- How?
 - **release resources from a process** – if it is possible
 - The process is suspended until the resource can be returned back
 - Efficient but requires the possibility of saving the process state
 - **rollback** – if the states of execution of the different processes is periodically saved
 - A resource is released from a process, whose state of execution is rolled back to the time the resource was assigned to it
 - **kill processes**
 - Radical but an easy to implement method

Bibliography

- Operating Systems: Internals and Design Principles, W. Stallings, Prentice-Hall International Editions, 7th Ed, 2012
 - Chapter 6: Concurrency: deadlock and starvation (sections 6.1 to 6.7)
- Operating Systems Concepts, A. Silberschatz, P. Galvin and G. Gagne, John Wiley & Sons, 9th Ed, 2013
 - Chapter 7: Deadlocks (sections 7.1 to 7.6)
- Modern Operating Systems, A. Tanenbaum and H. Bos, Pearson Education Limited, 4th Ed, 2015
 - Chapter 6: Deadlocks (section 6.1 to 6.6)



Sistemas de Operação / Fundamentos de Sistemas Operativos

Memory management

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

Outline

- ① Introduction
- ② Address space of a process
- ③ Contiguous memory allocation
- ④ Memory partitioning
- ⑤ Virtual memory system
- ⑥ Paging
- ⑦ Segmentation
- ⑧ Page replacement
- ⑨ Bibliography

Memory management

Introduction

- To be executed, a process must have its address space, at least partially, resident in main memory
- In a multiprogrammed environment, to maximize processor utilization and improve response time (or turnaround time), a computer system must maintain the address spaces of multiple processes resident in main memory
- But, there may not be room for all
 - because, although the main memory has been growing over the years, it is a fact that “data expands to fill the space available for storage”

(Corollary of the Parkinson's law)

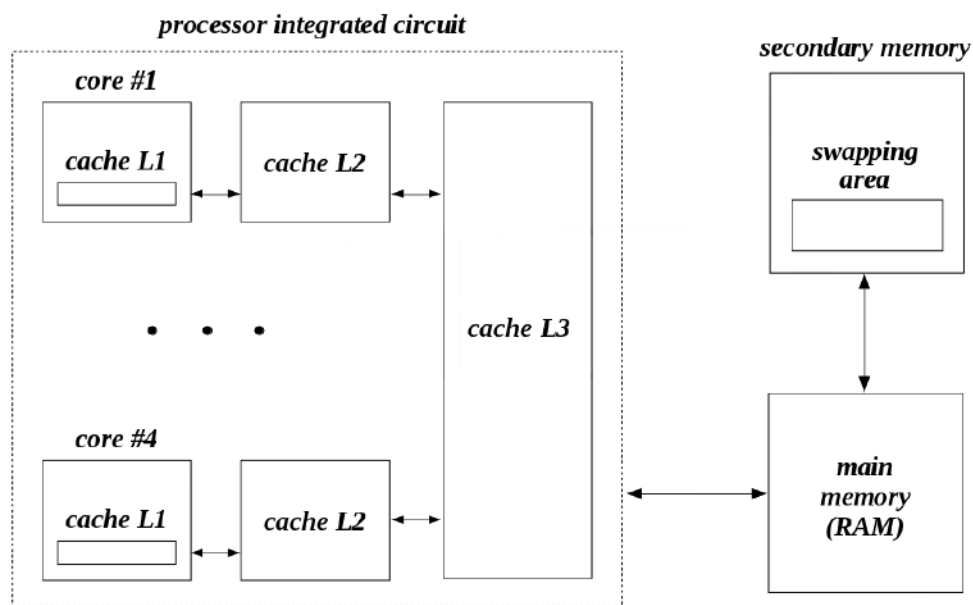
Memory management

Memory hierarchy

- Ideally, an application programmer would like to have infinitely large, infinitely fast, non-volatile and inexpensive available memory
 - In practice, this is not possible
- Thus, the memory of a computer system is typically organized at different levels, forming a hierarchy
 - **cache memory** – small (tens of KB to some MB), very fast, volatile and expensive
 - **main memory** – medium size (hundreds of MB to hundreds of GB), volatile and medium price and medium access speed
 - **secondary memory** – large (tens, hundreds or thousands of GB), slow, non-volatile and cheap

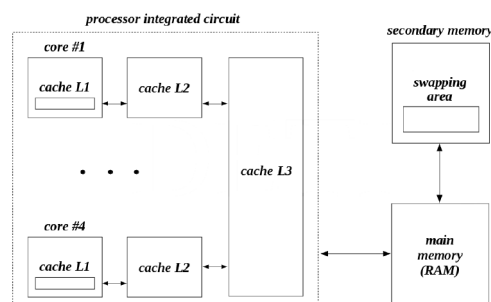
Memory management

Memory hierarchy (2)



Memory management

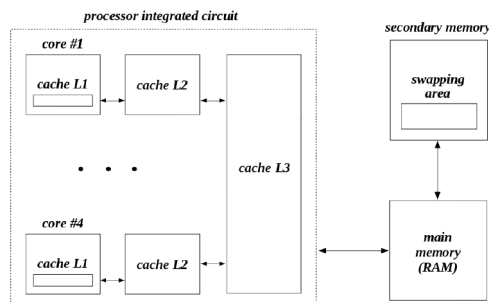
Memory hierarchy (3)



- The **cache memory** will contain a copy of the memory positions (instructions and operands) most frequently referenced by the processor in the near past
 - The cache memory is located on the processor's own integrated circuit (**level 1**)
 - And on an autonomous integrated circuit glued to the same substrate (**levels 2 and 3**)
- Data transfer to and from main memory is done almost completely transparent to the system programmer

Memory management

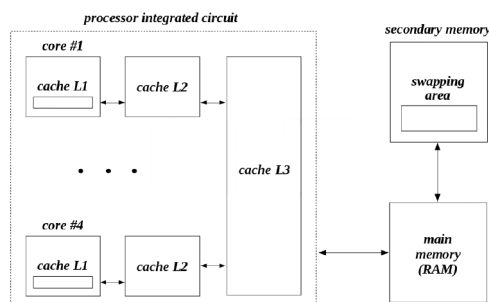
Memory hierarchy (3)



- **Secondary memory** has two main functions
 - **File system** – storage for more or less permanent information (programs and data)
 - **Swapping area** – Extension of the main memory so that its size does not constitute a limiting factor to the number of processes that may currently coexist
 - the swapping area can be on a disk partition used only for that purpose or be a file in a file system

Memory management

Memory hierarchy (4)



- This type of organization is based on the assumption that the further an instruction or operand is away from the processor, the less times it will be referenced
 - In these conditions, the mean time for a reference tends to be close to the lowest value
- Based on the **principle of locality of reference**
 - The tendency of a program to access the same set of memory locations repetitively over a short period of time

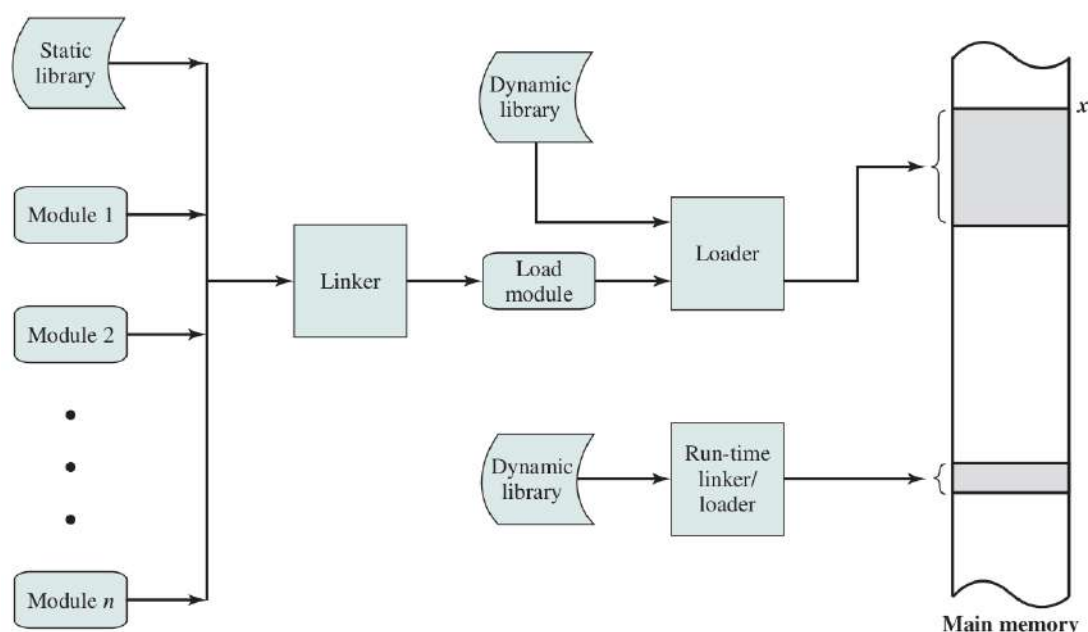
Memory management

Role

- The role of memory management in a multiprogramming environment focuses on allocating memory to processes and on controlling the transfer of data between main and secondary memory (**swapping area**), in order to
 - **Maintaining a register** of the parts of the main memory that are occupied and those that are free
 - **Reserving portions** of main memory for the processes that will need it, or **releasing** them when they are no longer needed
 - **Swapping out** all or part of the address space of a process when the main memory is too small to contain all the processes that coexist.
 - **Swapping in** all or part of the address space of a process when main memory becomes available

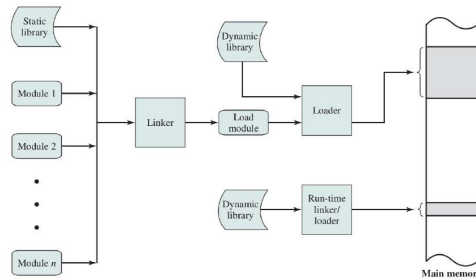
Address space

Linker and loader roles



Address space

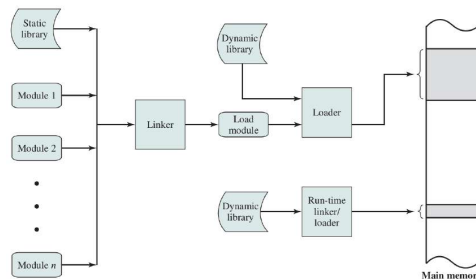
Linker and loader roles (2)



- The **object files**, resulting from the compilation process, are relocatable files
 - The addresses of the various instructions, constants and variables are calculated from the beginning of the module, by convention the address 0
- The role of the **linking process** is to bring the different object files together into a single file, the **executable file**, resolving among themselves the various external references
 - **Static libraries** are also included in the linking process
 - **Dynamic (shared) libraries** are not

Address space

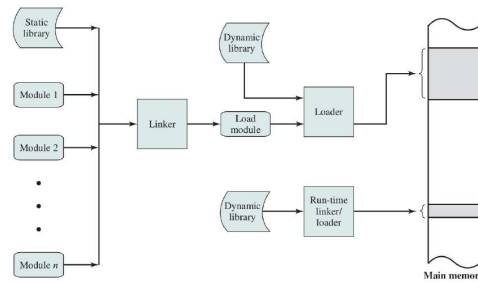
Linker and loader roles (3)



- The **loader** builds the binary image of the process **address space**, which will eventually be executed, combining the executable file and, if applicable, some **dynamic libraries**, resolving any remaining external references
- Dynamic libraries can also only be loaded at run time

Address space

Linker and loader roles (4)



- When the linkage is dynamic
 - Each reference in the code to a routine of a dynamic library is replaced by a **stub**
 - a small set of instructions that determines the location of a specific routine, if it is already resident in main memory, or promotes its load in memory, otherwise
 - When a stub is executed, the associated routine is identified and located in main memory, the stub then replaces the reference to its address in the process code with the address of the system routine and executes it
 - When that code zone is reached again, the system routine is now executed directly
- All processes that use the same dynamic library, execute the same copy of the code, thus minimizing the main memory occupation

Address space

Object and executable files

source file

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

int main (void)
{
    printf ("hello, world!\n");
    exit (EXIT_SUCCESS);
}
```

object file

```
$ gcc -Wall -c hello.c

$ file hello.o
hello.o: ELF 64-bit LSB relocatable,
x86-64, version 1 (SYSV), not stripped
```

executable file

```
$ gcc -o hello hello.o

$ file hello
hello: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=48ac0a8ba08d8df6d5e8a27e00b50248a3061876, not stripped
```

Address space

Object and executable files (2)

```
$ objdump -fstr hello.o
hello.o:      file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000
SYMBOL TABLE:
0000000000000000 1      df *ABS*          0000000000000000 z.c
0000000000000000 1      d  .text          0000000000000000 .text
0000000000000000 1      d  .data          0000000000000000 .data
0000000000000000 1      d  .bss 0000000000000000 .bss
0000000000000000 1      d  .rodata        0000000000000000 .rodata
0000000000000000 1      d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 1      d  .eh_frame        0000000000000000 .eh_frame
0000000000000000 1      d  .comment        0000000000000000 .comment
0000000000000000 g      F  .text          000000000000001a main
0000000000000000      *UND*          0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000      *UND*          0000000000000000 puts
0000000000000000      *UND*          0000000000000000 exit

...
```

Address space

Object and executable files (3)

```
RELOCATION RECORDS FOR [.text]:
OFFSET          TYPE          VALUE
0000000000000007 R_X86_64_PC32      .rodata-0x0000000000000004
000000000000000c R_X86_64_PLT32     puts-0x0000000000000004
0000000000000016 R_X86_64_PLT32     exit-0x0000000000000004

RELOCATION RECORDS FOR [.eh_frame]:
OFFSET          TYPE          VALUE
0000000000000020 R_X86_64_PC32      .text

Contents of section .text:
0000 554889e5 488d3d00 000000e8 00000000  UH..H.=.....
0010 bf000000 00e80000 0000          .....
Contents of section .rodata:
0000 68656c6c 6f2c2077 6f726c64 2100      hello, world!.
Contents of section .comment:
0000 00474343 3a202855 62756e74 7520372e  .GCC: (Ubuntu 7.
0010 332e302d 32377562 756e7475 317e3138  3.0-27ubuntu1~18
0020 2e303429 20372e33 2e3000      .04) 7.3.0.
Contents of section .eh_frame:
0000 14000000 00000000 017a5200 01781001  ....zR..x..
0010 1b0c0708 90010000 1c000000 1c000000  ....
0020 00000000 1a000000 00410e10 8602430d  ....A....C.
0030 06000000 00000000          .....
```

Address space

Object and executable files (4)

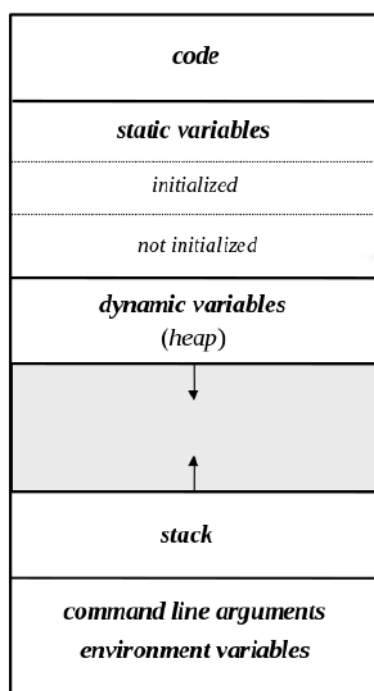
```
$ objdump -fTR hello
z:      file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000000000000580

DYNAMIC SYMBOL TABLE:
0000000000000000 w D *UND*      0000000000000000
_ITM_deregisterTMCloneTable
0000000000000000 DF *UND*      0000000000000000 GLIBC_2.2.5 puts
0000000000000000 DF *UND*      0000000000000000 GLIBC_2.2.5 __libc_start_main
0000000000000000 w D *UND*      0000000000000000 __gmon_start__
0000000000000000 DF *UND*      0000000000000000 GLIBC_2.2.5 exit
0000000000000000 w D *UND*      0000000000000000
_ITM_registerTMCloneTable
0000000000000000 w DF *UND*      0000000000000000 GLIBC_2.2.5 __cxa_finalize

DYNAMIC RELOCATION RECORDS
OFFSET      TYPE      VALUE
0000000000200db0 R_X86_64_RELATIVE *ABS*+0x00000000000000680
0000000000200db8 R_X86_64_RELATIVE *ABS*+0x00000000000000640
0000000000201008 R_X86_64_RELATIVE *ABS*+0x000000000000201008
0000000000200fd8 R_X86_64_GLOB_DAT _ITM_deregisterTMCloneTable
0000000000200fe0 R_X86_64_GLOB_DAT __libc_start_main@GLIBC_2.2.5
0000000000200fe8 R_X86_64_GLOB_DAT __gmon_start__
0000000000200ff0 R_X86_64_GLOB_DAT _ITM_registerTMCloneTable
0000000000200ff8 R_X86_64_GLOB_DAT __cxa_finalize@GLIBC_2.2.5
0000000000200fc8 R_X86_64_JUMP_SLOT puts@GLIBC_2.2.5
0000000000200fd0 R_X86_64_JUMP_SLOT exit@GLIBC_2.2.5
```

Address space

Address space of a process



- **Code** and **static variables** regions have a fixed size, which is determined by the loader
- **Dynamic variables** and **stack** regions grow (in opposite directions) during the execution of the process
- It is a common practice to leave an unallocated memory area in the process address space between the dynamic definition region and the stack that can be used alternatively by any of them
- When this area is exhausted on the stack side, the execution of the process cannot continue, resulting in the occurrence of a fatal error: **stack overflow**

Address space

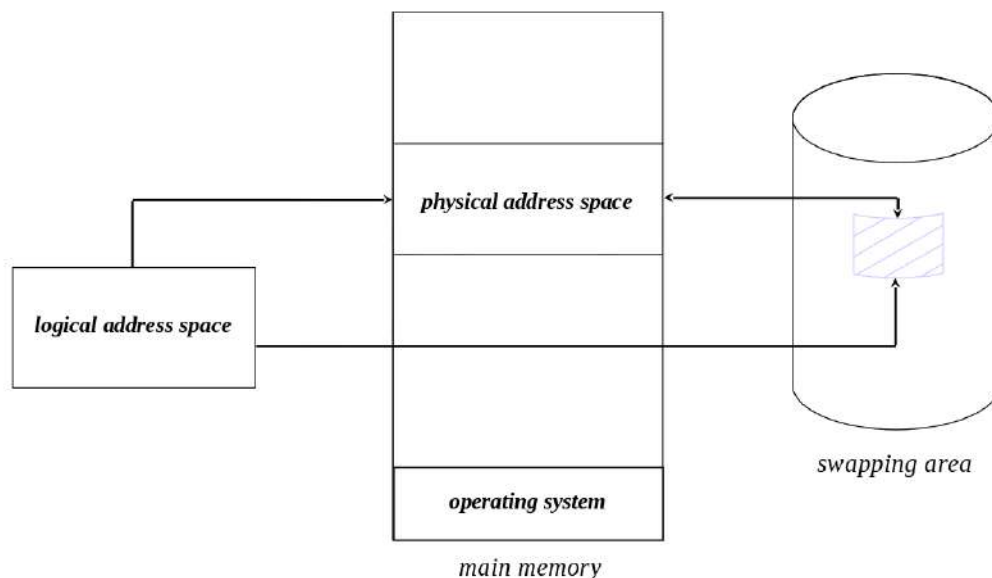
Address space of a process

- The binary image of the process address space represents a relocatable address space, the so-called **logical address space**
- The main memory region where it is loaded for execution, constitutes the **physical address space** of the process
- Separation between the logical and physical address spaces is a central concept to the memory management mechanisms in a multiprogrammed environment
- There are two issues that have to be solved
 - **dynamic mapping** – ability to convert a logical address to a physical address at runtime, so that the physical address space of a process can be placed in any region of main memory and be moved if necessary
 - **dynamic protection** – ability to prevent at runtime access to addresses located outside the process's own address space

Contiguous memory allocation

Logical and physical address spaces

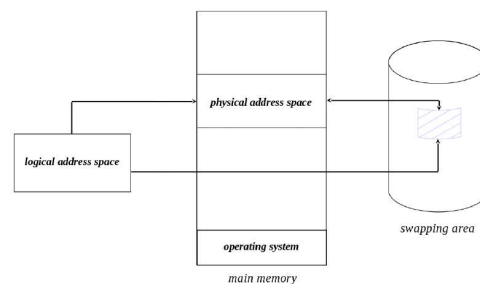
- In contiguous memory allocation, there is a **one-to-one correspondence** between the **logical address space** of a process and its **physical address space**



Contiguous memory allocation

Logical and physical address spaces

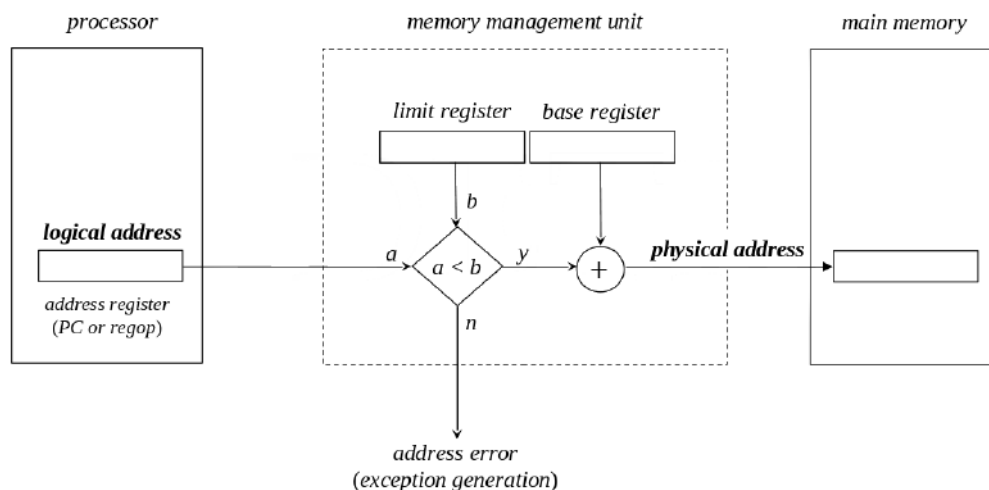
- Consequences:
 - **Limitation of the address space of a process** – in no case can memory management support automatic mechanisms that allow the address space of a process to be larger than the size of the main memory available
 - The use of overlays can allow to overcome that
 - **Contiguity of the physical address space** – although it is not a strictly necessary condition, it is naturally simpler and more efficient to assume that the process address space is contiguous
 - **Swapping area as an extension of the main memory** – it serves to storage the address space of processes that cannot be resident into main memory due to lack of space



Contiguous memory allocation

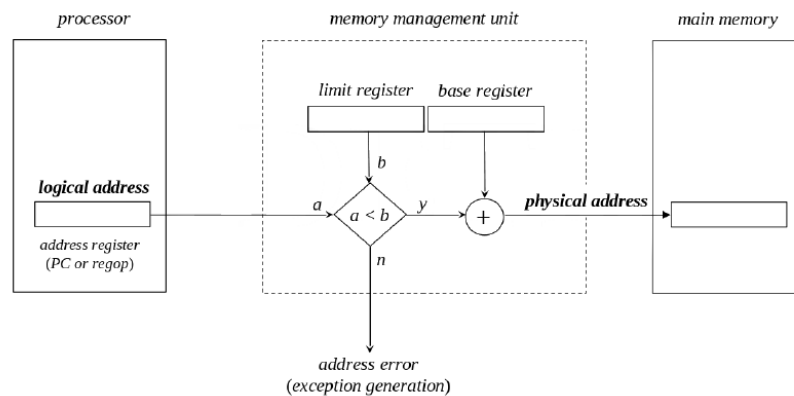
Logical address to physical address translation

- How are **dynamic mapping** and **dynamic protection** accomplished?
 - A piece of hardware (the MMU) is required



Contiguous memory allocation

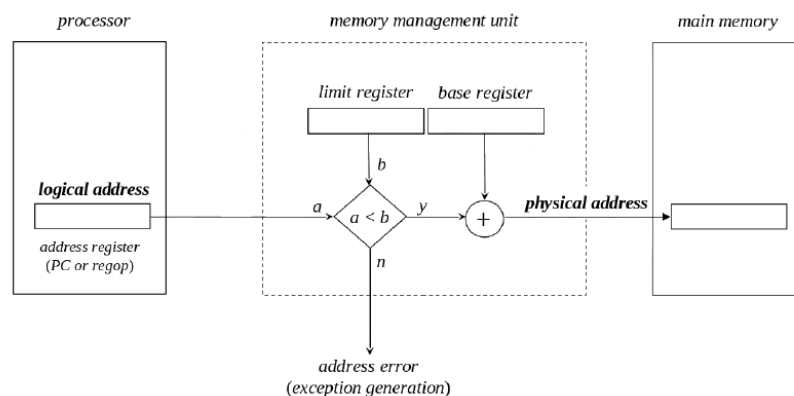
Logical address to physical address translation (2)



- The **limit register** must contain the size in bytes of the logical address space
- The **base register** must contain the address of the beginning of the main memory region where the physical address space of the process is placed
- On context switching, the **dispatch** operation loads the base and limit registers with the values present in the corresponding fields of the process control table entry associated with the process that is being scheduled for execution

Contiguous memory allocation

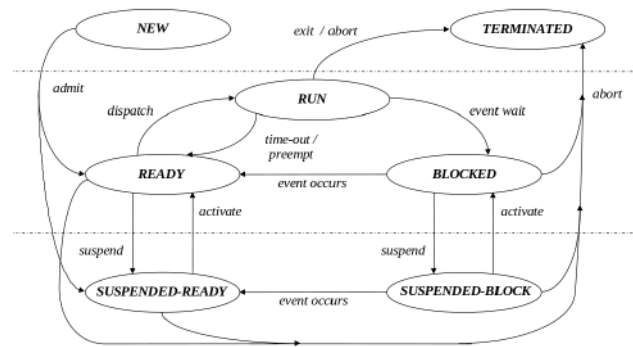
Logical address to physical address translation (2)



- Whenever there is a reference to memory
 - the logical address is first compared to the value of the limit register
 - if it is less, it is a valid reference (it occurs within the process address space) then the logical address is added to the value of the base register to produce the physical address
 - if it is greater than or equal, it is an invalid reference, then a null memory access (dummy cycle) is set in motion and an exception is generated due to address error

Contiguous memory allocation

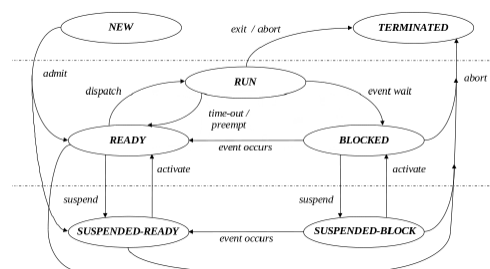
Long-term scheduling



- When a process is created, the data structures to manage it is initialized
 - Its logical address space is constructed, and the value of the limit register is computed and saved in the corresponding field of the process control table (PCT)
- If there is space in main memory, its address space is loaded there, the base register field is updated with the initial address of the assigned region and the process is placed in the READY queue
- Otherwise, its address space is temporarily stored in the swapping area and the process is placed in the SUSPENDED-READY queue

Contiguous memory allocation

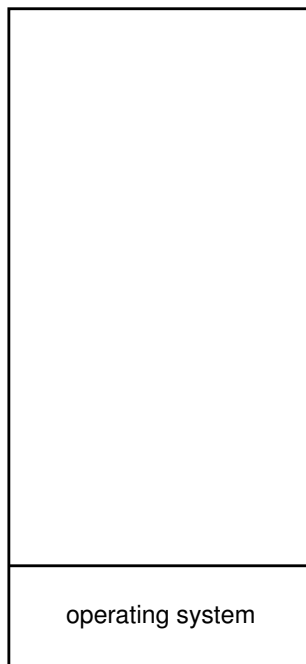
Medium-term scheduling



- If memory is required for another process, a BLOCKED (or even READY) process may be **swapped out**, freeing the physical memory it is using,
 - In such a case, its base register field in the PCT becomes undefined
- If memory becomes available, a SUSPENDED-READY (or even SUSPENDED-BLOCKED) process may be **swapped in**,
 - Its base register field in the PCT is updated with its new physical location
 - A SUSPENDED-BLOCK process is only selected if no SUSPENDED-READY one exists
- When a process terminates, it is **swapped out** (if not already there), waiting for the end of operations

Memory partitioning

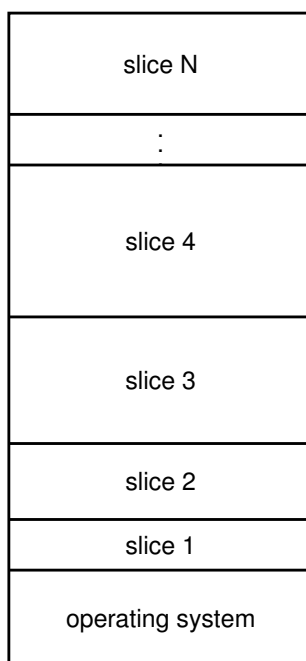
How to do it?



- After reserving some amount to the operating system, how to partition the real memory to accommodate the different processes?
- Fix partition
 - into slices of equal size
 - into slices of different size
- Dynamic partition
 - being done as being required?

Memory partitioning

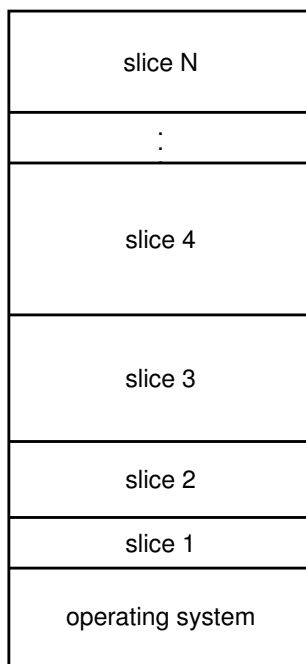
Fixed partitioning



- Main memory can be divided into a number of static slices at system generation time
 - not necessarily all the same size
- The logical address space of a process may be loaded into a slice of equal or greater size
 - thus, the largest slice determines the size of the largest allowable process
- Some features:
 - Simple to implement
 - Efficient – little operating system overhead
 - Fixed number of allowable processes
 - Inefficient use of memory due to internal fragmentation – the part of a slice not used by a process is wasted

Memory partitioning

Fixed partitioning (2)

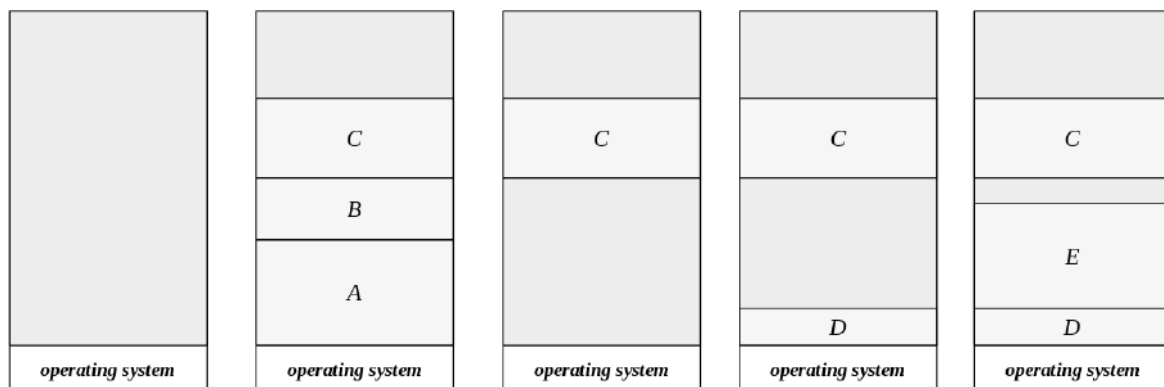


- If a slice becomes available, which of the SUSPENDED-READY processes should be placed there?
- Two different scheduling policies are here considered
 - **Valuing fairness** – the first process in the queue of SUSPENDED-READY processes whose address space fits in the slice is chosen
 - **Valuing the occupation of main memory** – the first process in the queue of SUSPENDED-READY processes with the largest address space that fits in the slice is chosen
 - to avoid starvation an aging mechanism can be used

Memory partitioning

Dynamic partitioning

- In dynamic partitioning, at start, all the available part of the memory constitutes a single block and then
 - reserve a region of sufficient size to load the address space of the processes that arises
 - release that region when it is no longer needed



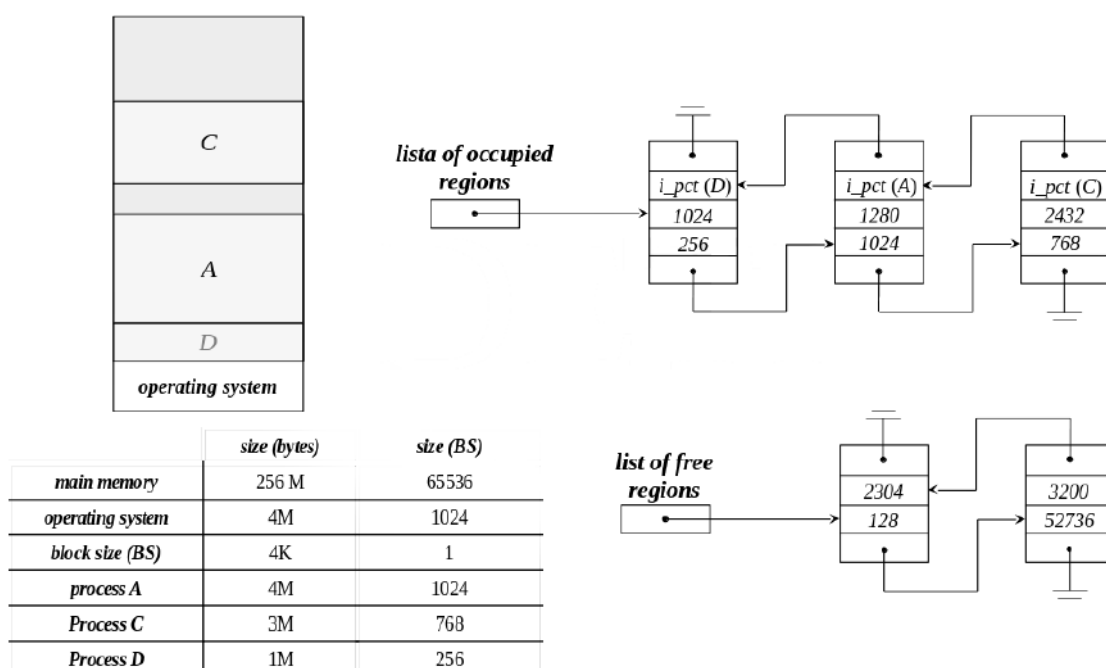
Memory partitioning

Dynamic partitioning (2)

- As the memory is dynamically reserved and released, the operating system has to keep an updated record of occupied and free regions
- One way to do this is by building two (bi)linked lists
 - **list of occupied regions** – locates the regions that have been reserved for storage of the address spaces of processes resident in main memory
 - **list of free regions** – locates the regions still available
- Memory is not allocated in byte boundaries, because
 - useless, very small free regions may appear
 - that will be included in the list of free regions
 - making subsequent searches more complex
- Thus, the main memory is typically divided into blocks of fixed size and allocation is made in units of these blocks

Memory partitioning

Dynamic partitioning (3)



Memory partitioning

Dynamic partitioning (4)

- **Valuing fairness** is the scheduling discipline generally adopted, being chosen the first process in the queue of SUSPENDED-READY processes whose address space can be placed in main memory
- Dynamic partitioning can produce **external fragmentation**
 - Free space is splitted in a large number of (possible) small free regions
 - Situations can be reached where, although there is enough free memory, it is not continuous and the storage of the address space of a new or suspended process is no longer possible
- The solution is **garbage collection** – compact the free space, grouping all the free regions into a single one
 - This operation requires stopping all processing and, if the memory is large, can have a very long execution time

Memory partitioning

Dynamic partitioning (5)

- In case there are several free regions available, which one to use to allocate the address space of a process?
- Possible policies:
 - **first fit** – the list of free regions is searched from the beginning until the first region with sufficient size is found
 - **next fit** – is a variant of the first fit which consists of starting the search from the stop point in the previous search
 - **best fit** – the list of free regions is fully searched, choosing the smallest region with sufficient size for the process
 - **worst fit** – the list of free regions is fully searched, choosing the largest existing region
- Which one is the best?
 - in terms of fragmentation
 - in terms of efficiency of allocation
 - in terms of efficiency of release

Memory partitioning

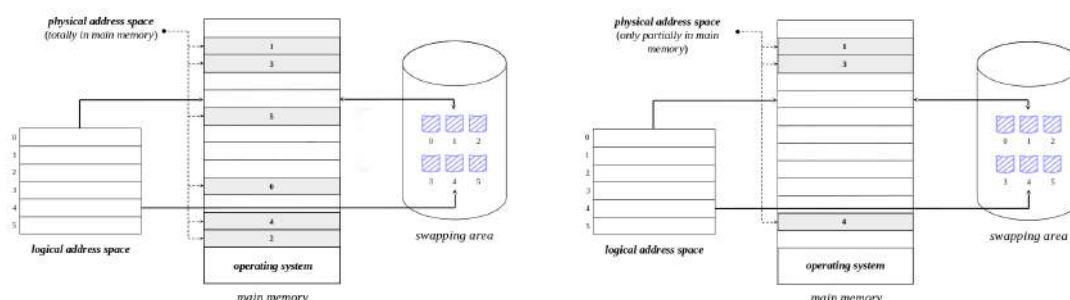
Dynamic partitioning (6)

- Advantages
 - **general** – the scope of application is independent of the type of processes that will be executed
 - **low complexity implementation** – no special hardware required and data structures are reduced to two (bi)linked lists
- Disadvantages
 - **external fragmentation** – the fraction of the main memory that ends up being wasted, given the small size of the regions in which it is divided, can reach in some cases about a third of the total (50% rule)
 - **inefficient** – it is not possible to build algorithms that are simultaneously very efficient in allocating and freeing space

Virtual memory system

Mapping of the logical address space

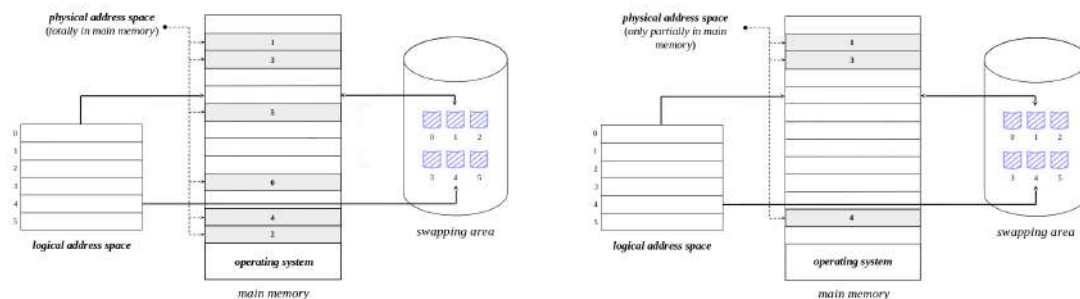
- In a virtual memory system, the **logical address space** of a process and its **physical address space** are **totally dissociated**
- The **logical address space** is sliced in different blocks
- The different blocks are allocated independently of each other
 - So they can spread along the physical address space
- A process can be only partially resident in main memory
 - So some of its blocks may be only in the swapping area



Virtual memory system

Some features

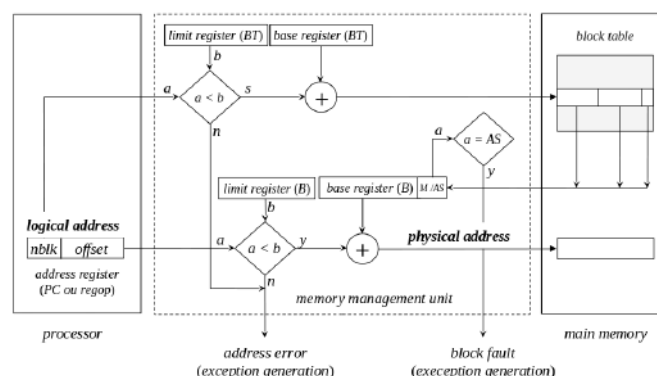
- **Non-contiguity of the physical address space** – the address spaces of the processes, divided into blocks of fixed or variable size, are dispersed throughout the memory, trying to guarantee a more efficient occupation of the available space
- **No limitation of the address space of a process** – methodologies allowing the execution of processes whose address spaces are greater than the size of the available main memory can be established
- **Swapping area as an extension of the main memory** – its role is to maintain an updated image of the address spaces of the processes that currently coexist, namely their variable part (static and dynamic definition areas and stack)



Virtual memory system

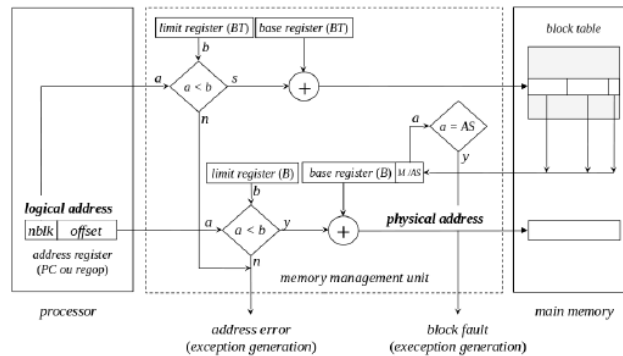
Logical address to physical address translation

- How are **dynamic mapping** and **dynamic protection** accomplished?
- A **logical address** is composed of two parts:
 - **nblk** – that identifies a specific logical block
 - **offset** – that identifies a position within the block, as an offset from its beginning
- A **block table**, stored in memory, maps every logical block to its physical counterpart
- The **MMU** must deal with this structure



Virtual memory system

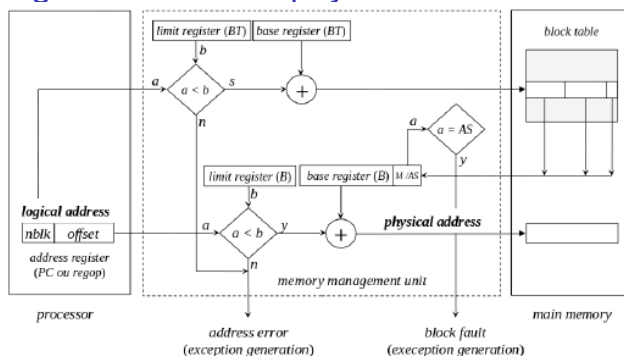
Logical address to physical address translation (2)



- The MMU must contain two pairs of base and limit registers
 - one pair (BT) is related to the beginning and size of the **block table**, data structure describing the several blocks the logical address space is divided in
 - the other pair (B) describes a specific block, the one that is being accessed
- On context switching, the **dispatch** operation loads those related to the block table, with the values stored in the PCT entry of the process scheduled for execution
 - the **BT base register** represents the start address of the process' block table
 - the **BT limit register** represents the number of table entries (number of blocks)

Virtual memory system

Logical address to physical address translation (3)



- A memory access unfolds into three steps
- Step 1:
 - field **nblk** of the logical address is compared with **BT limit register**
 - if it is valid (\leq), the **BT base register** plus **nblk** points to the **block table entry**, which is loaded into the MMU
 - if not ($>$), a null memory access (dummy cycle) is set in motion and an **exception** is generated due to **address error**

- Step 2:
 - flag **M/AS** is evaluated
 - if it is M (the block being referenced is in memory), the operation may proceed
 - if not (the block being referenced is swapped out), a null memory access (dummy cycle) is set in motion and an **exception** is generated due to **block fault**
- Step 3:
 - field **offset** of the logical address is compared with **B limit register**
 - if it is valid (\leq), the **B base register** plus **offset** points to the **physical address**
 - if not ($>$), a null memory access is set in motion and an **exception** is generated due to **address error**

- A memory access unfolds into three steps
- Step 1:

Virtual memory system

Analysis

- The increase in versatility, introduced by the virtual memory system, has the cost of transforming each memory access into two accesses
 - in the first, the process' block table is accessed, to obtain the address of the beginning of the block in memory
 - only in the second the specific memory position is accessed
- Conceptually, the virtual memory system results in a partitioning of the logical address space of the process in blocks that are dynamically treated as autonomous address sub-spaces in a contiguous memory allocation
 - of fixed partitions, if the blocks are all the same size
 - of variable partitions, if they can have different sizes
- What's new is the possibility to access a block currently not resident in main memory
 - with the consequent need to cancel the access and repeat it later, when the block is already loaded

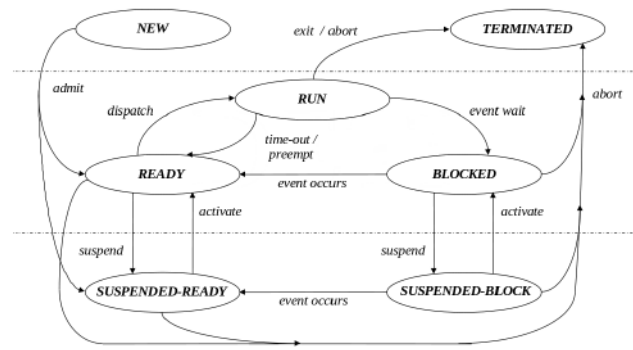
Virtual memory system

Role of the TLB

- The need for this double access to memory can be minimized by taking advantage of the [principle of locality of reference](#)
- As the accesses will tend to be concentrated in a well-defined set of blocks during extended process execution time intervals, the memory management unit (MMU) usually keeps the content of the block table entries stored in an internal associative memory, called the [translation lookaside buffer \(TLB\)](#)
- Thus, the first access can be a
 - [hit](#) – when the entry is stored in the TLB, in which case the access is internal to the MMU
 - [miss](#) – when the entry is not stored in the TLB, in which case there is access to the main memory
- The average access time to an instruction or operand tends to approximate the lowest value
 - an access to the TLB plus an access to the main memory

Virtual memory system

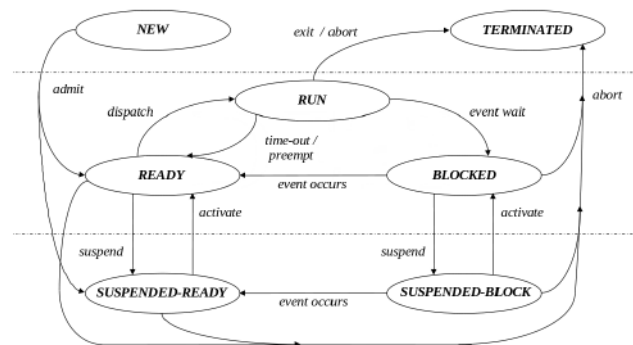
Long-term scheduling



- When a process is created, the data structures to manage it is initialized
 - Its logical address space is constructed, at least its variable part is put in the swapping area and its block table is organized
 - Some blocks can be shared with other processes
- If there is space in main memory, at least its block table, first block of code and block of the its stack are loaded there, the corresponding entries in the block table are updated and the process is placed in the READY queue
- Otherwise, the process is placed in the SUSPENDED-READY queue

Virtual memory system

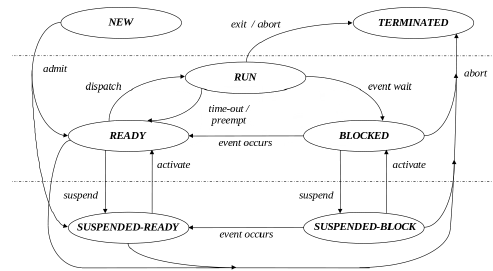
Short-term scheduling



- During its execution, on **block fault**, a process is placed in the BLOCKED state, while the **faulty block is swapped in**
- When the block is in memory, the process is placed in the READY state

Virtual memory system

Medium-term scheduling



- While READY or BLOCKED, all blocks of a process may be **swapped out**, if memory space is required
- While SUSPENDED-READY (or SUSPENDED-BLOCKED), if memory space becomes available, the block table and a selection of blocks of a process may be **swapped in**, and the process transitions to READY or BLOCKED
 - the corresponding entries of the block table are updated
 - A SUSPENDED-BLOCK process is only selected if no SUSPENDED-READY one exists
- When a process terminates, it may be **swapped out** (if not already there), waiting for the end of operations

Virtual memory system

Block fault exception

- A relevant property of the virtual memory system is the capacity of the computer system for executing processes whose address space is not in its entirety, and simultaneously, residing in main memory
- The operating system has to provide means to solve the problem of referencing an address located in a block that is not currently present in memory
- The MMU generates an exception in these circumstances and its service routine must start actions to **swap in** the block and, after its completion, **repeat** the execution of the instruction that produced the fault
- All these operations are carried out in a **completely transparent** way to the user who is not aware of the interruptions introduced in the execution of the process

Virtual memory system

Block fault exception – service procedure

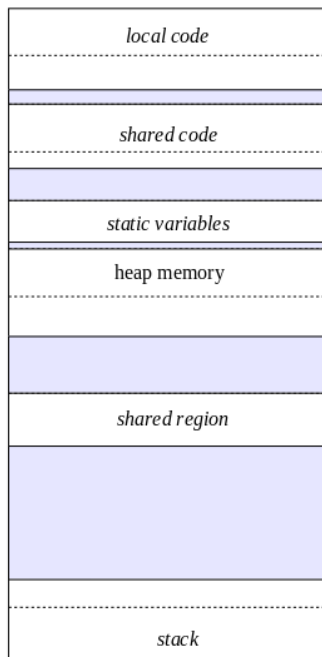
- The context of the process is saved in the corresponding entry of the process control table, its state changes to BLOCKED and the program counter is updated to the address that produced the block fault
- If memory space is available, a region is selected to swap in the missing block
- If not, an occupied block is selected to be replaced
 - if this block was modified, it is transferred to the swapping area
 - its entry in the block table is updated to indicate it is no longer in memory
- The swapping in of the missing block is started (to the selected region)
- The dispatch function of the processor scheduler is called to put in execution one of the READY process
- When the transfer is concluded, the block table entry is updated and the process is put in the READY state

Virtual memory system

Block fault exception – analysis

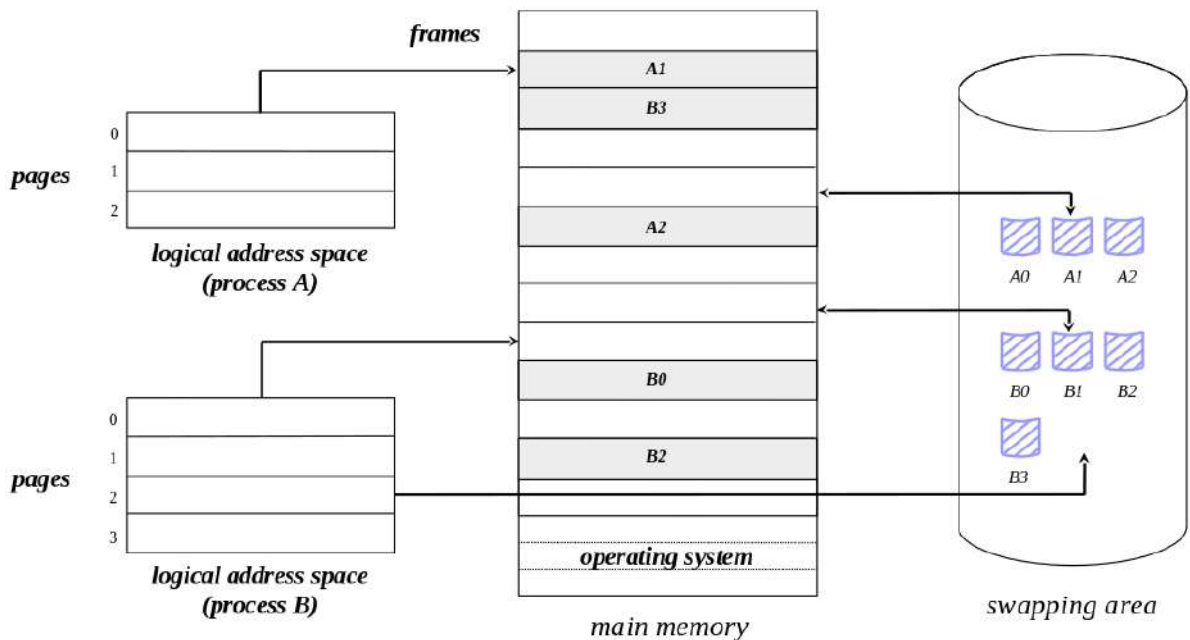
- If, during its execution, a process was continuously generating block fault exceptions, the processing rate would be very slow and, in general, the throughput of the computer system would also be very low, jeopardizing the usefulness of a organization of virtual memory in multiprogramming
- In practice, this is not the case, because of the principle of **locality of reference**
 - A process, whatever the stage of its execution, accesses during relatively extended periods of time only a small fraction of its logical address space
- This fraction naturally changes over time, but in each interval considered there are typically thousands of accesses that are made on well-defined fractions of its address space.
 - As long as the corresponding blocks are in memory, the rhythm of execution can proceed without the occurrence of block faults

Paging Introduction



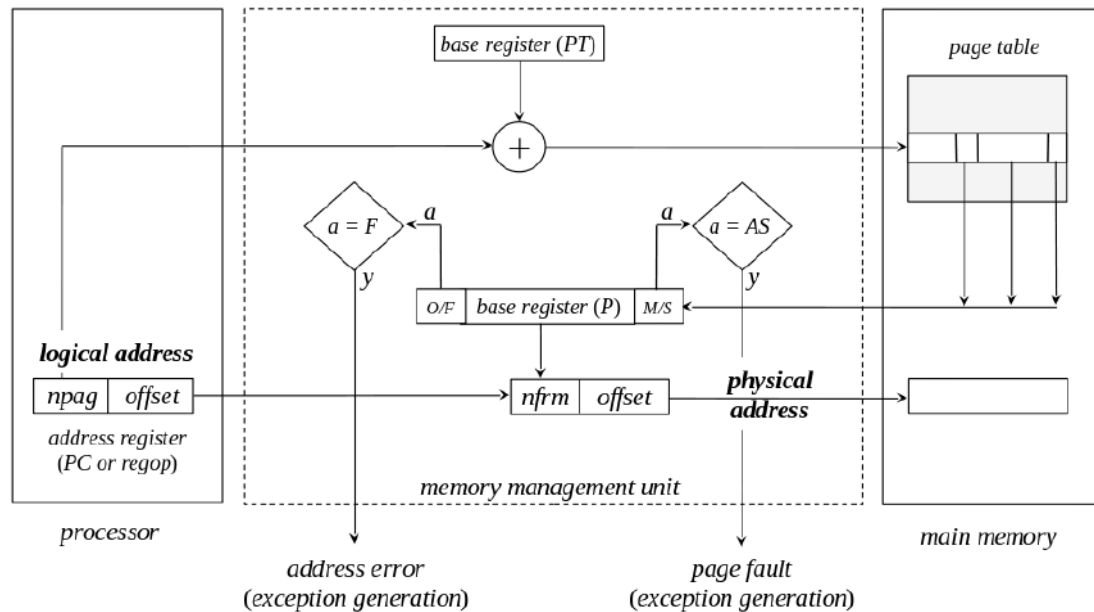
- Memory is divided into equal fixed-size chunks, called **frames**
 - a power of 2 is used for the size, typically 4 or 8 KB
- The logical address space of a process is divided into fixed-size blocks, of the same size, called **pages**
- While dividing the address space into pages, the linker usually starts a new page when a new segment starts
- In a logical address:
 - the most significant bits represent the **page number**
 - the least significant bits represent an **offset** within the page

Paging Illustration example



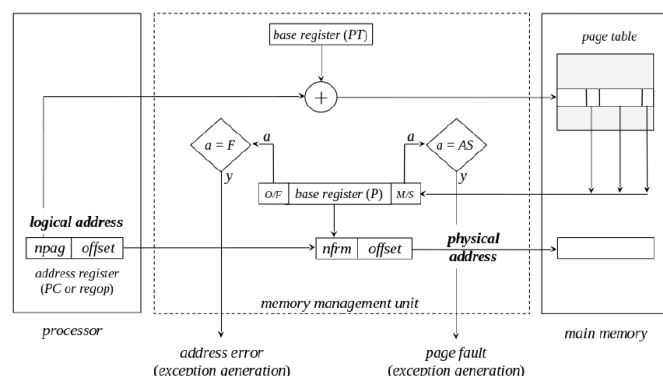
Paging

Memory management unit (MMU)



Paging

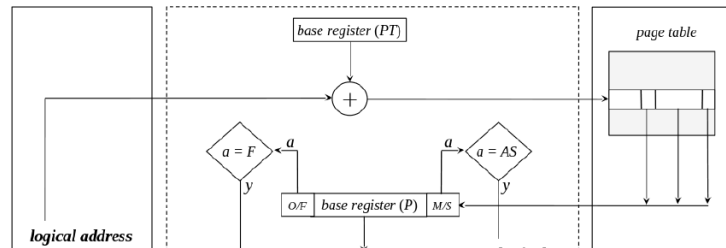
Logical address to physical address translation



- Structuring the logical address space of the process in order to map the whole, or at least a fraction, of the address space provided by the processor (in any case, always greater than or equal to the size of the existing main memory), it becomes possible to eliminate the need for the **limit register** associated with the size of the page table
 - As a consequence, the gap between the heap memory and the stack can be maximized
- There is not **limit register** associated with the page

Paging

Page table entries



- Page table contains one entry per page
- Entry definition:

O/F	M/S	ref	mod	perm	frame number	block number in swap area
-----	-----	-----	-----	------	--------------	---------------------------

- **O/F** – flag indicating if page has been already assigned to process
- **M/S** – flag indicating if page is in memory
- **ref** – flag indicating if page has been referenced
- **mod** – flag indicating if page has been modified
- **perm** – permissions
- **frame number** – frame where page is, if in memory
- **block number in swap area** – block where page is, in swapping area

Paging

Analysis

- Advantages:
 - **general** – the scope of your application is independent of the type of processes that will be executed (number and size of their address spaces)
 - **good usage of main memory** – does not lead to external fragmentation and internal fragmentation is practically negligible
 - **does not have special hardware requirements** – the memory management units in today's general-purpose processors implements it
- Disadvantages:
 - **longer memory access** – double access to memory, because of a prior access to the page table
 - Existence of a TLB (translation lookaside buffer) minimizes the impact
 - **very demanding operability** – requires the existence of a set of support operations, that are complex and have to be carefully designed to not compromise efficiency

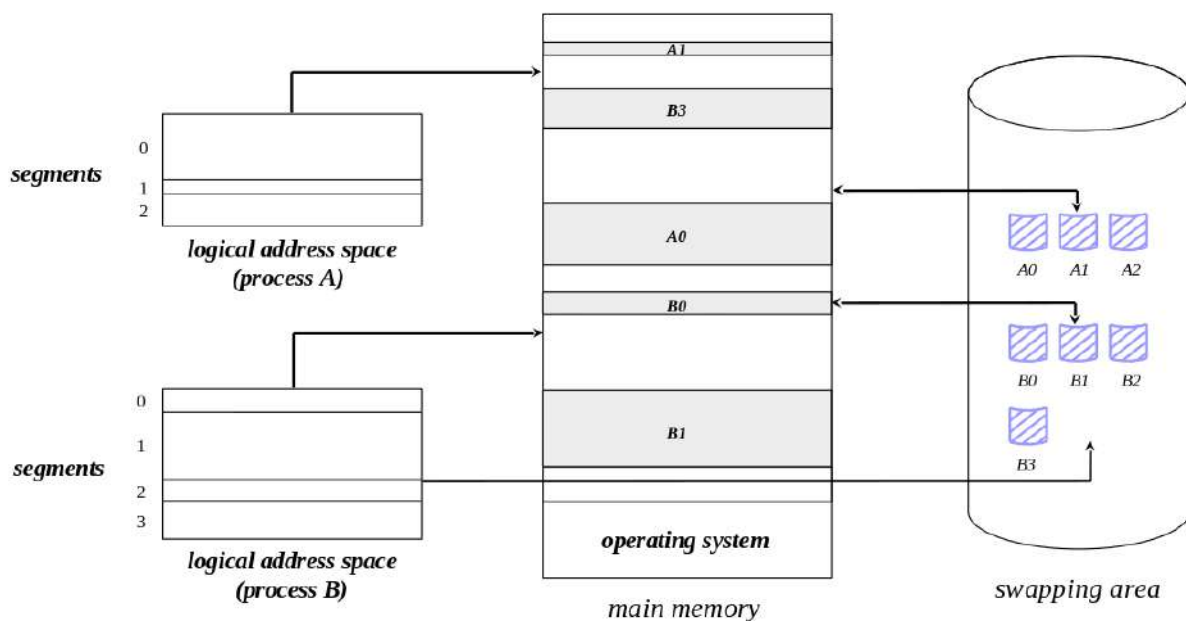
Segmentation

Introduction

- Typically, the logical address space of a process is composed of different type of segments:
 - **code** – one segment per code module
 - **static variables** – one segment per module containing static variables
 - **heap memory** – one segment
 - **shared memory** – one segment per shared region
 - **stack** – one segment
 - Different segments may have different sizes
- In a **segmentation architecture**, the segments of a process are manipulated separately
 - **Dynamic partitioning** may be used to allocate each segment
 - As a consequence, a process may not be contiguous in memory
 - Even, some segments may not be in main memory

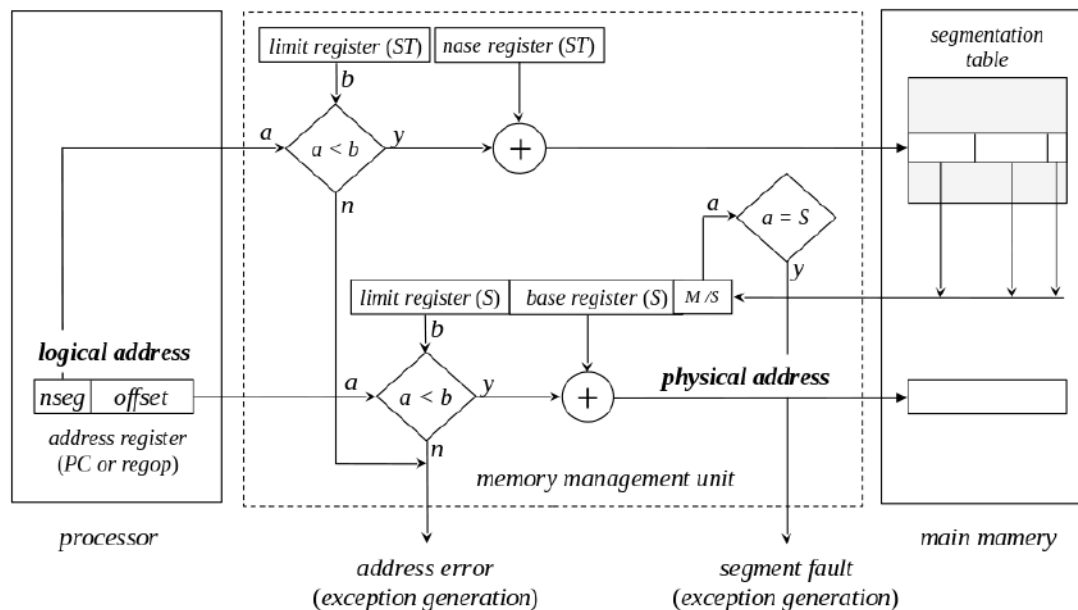
Segmentation

Illustration example



Segmentation

Memory management unit (MMU)



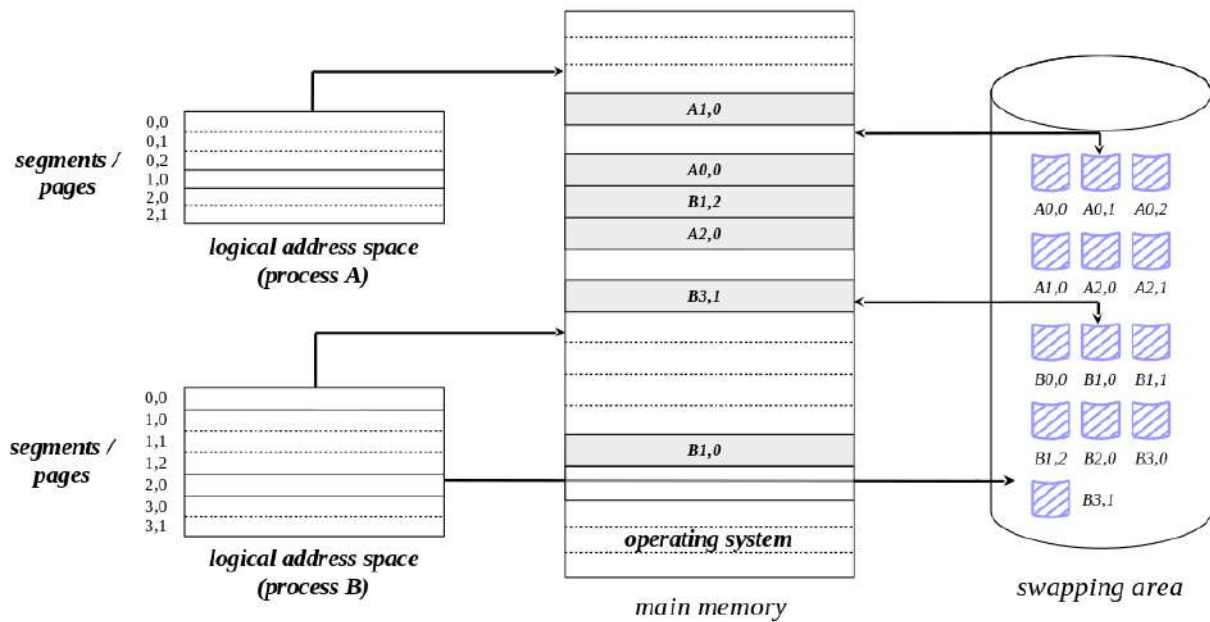
Combining segmentation and paging

Introduction

- Segmentation, taken alone, can have some drawbacks:
 - It may result in external fragmentation
 - A growing segment can impose a change in its location
- Merging segmentation and paging can solve these issues
 - First, the logical address space of a process is partitioned into segments
 - Then, each segment is divided into pages
- However, this introduces a growing complexity

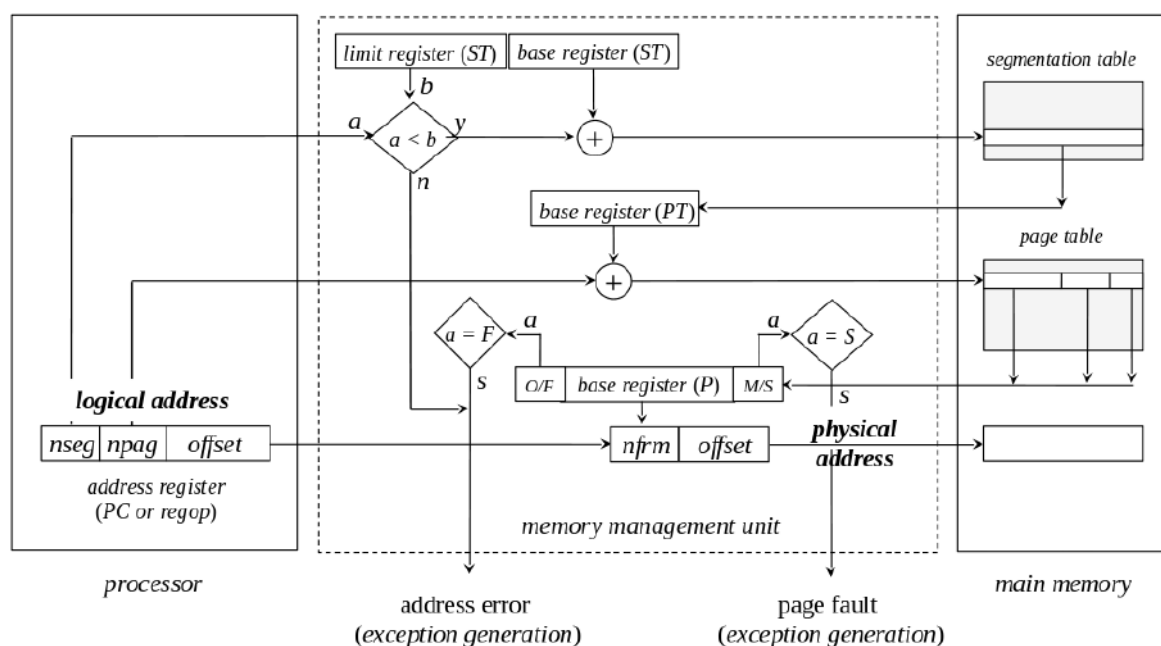
Combining segmentation and paging

Illustration example



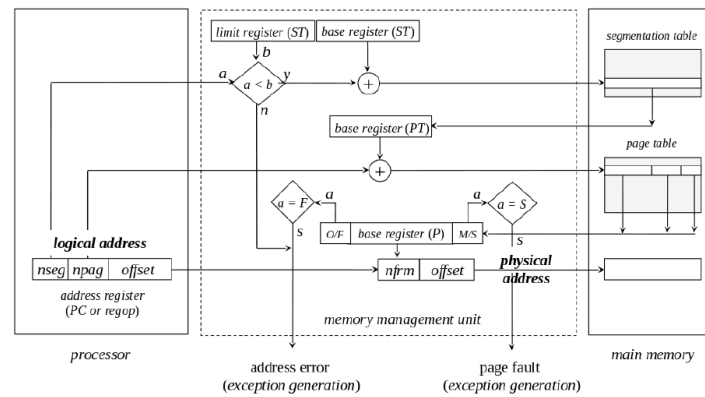
Combining segmentation and paging

Memory management unit (MMU)



Combining segmentation and paging

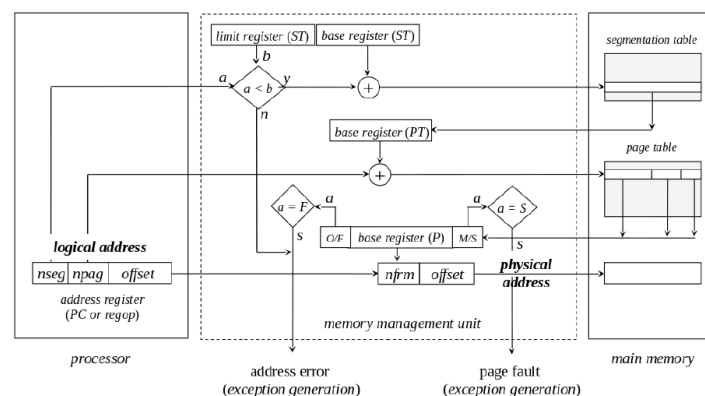
Logical address to physical address translation



- The MMU must contain 3 base registers and 1 limit register
 - 1 base register for the segmentation table
 - 1 limit register for the segmentation table
 - 1 base register for the page table
 - 1 base register for the memory frame

Combining segmentation and paging

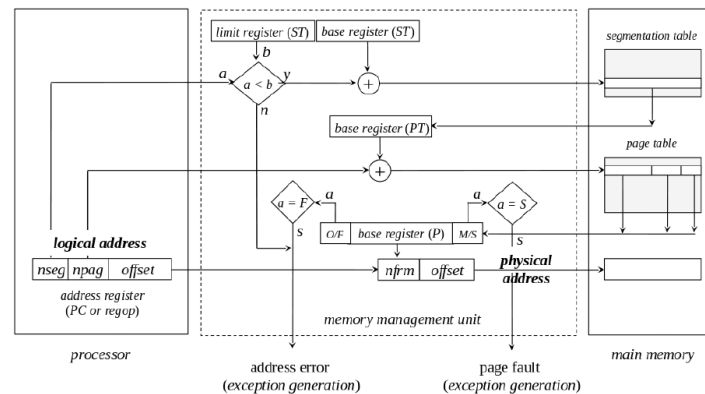
Logical address to physical address translation (2)



- An access to memory unfolds into 3 steps:
 - Access to the segmentation table
 - Access to the page table
 - Access to the physical address

Combining segmentation and paging

Logical address to physical address translation (3)



- Entry of the segmentation table

perm	memory address of the page table
------	----------------------------------

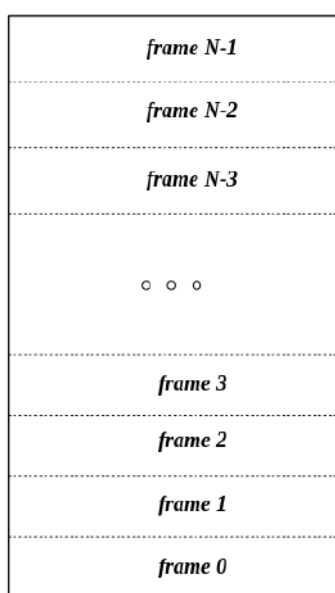
- Entry of the page table

O/F	M/S	ref	mod	frame number	block number in swap area
-----	-----	-----	-----	--------------	---------------------------

- The `perm` field is now associated to the segment

Page replacement

Introduction

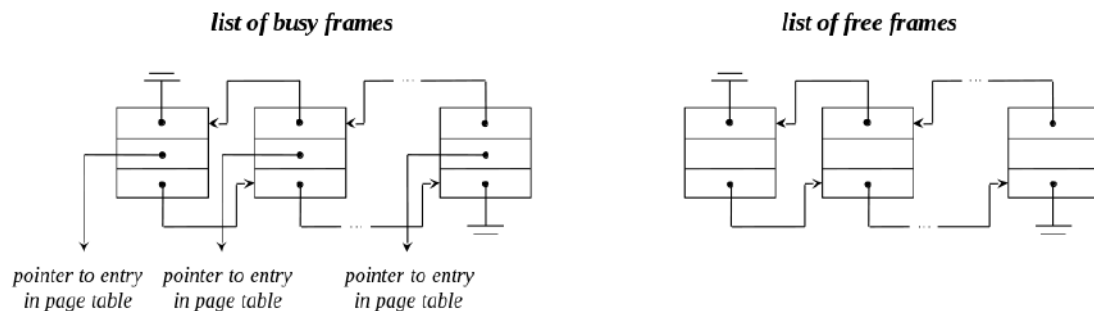


- In a paging (or combination of segmentation and paging) architecture, memory is partitioned into frames, each the same size as a page
 - A frame may be either **free** or **occupied** (containing a page)
- A page in memory may be:
 - locked** – if it can not be removed from memory (kernel, buffer cache, memory-mapped file)
 - unlocked** – if it can be removed from memory
- If no free frame is available, an occupied one may need to be released
 - This is the purpose of **page replacement**
- Page replacement only applies to unlocked pages

Page replacement

Lists of free and occupied frames

- Free frames are organized in a list – **list of free frames**
- Occupied frames, associated to unlocked pages, are also organized in a list – **list of occupied frames**
- How the list of occupied frames is organized depends on the **page replacement policy**



Page replacement

Action on page fault

- On **page fault**, if the list of free frames is empty, an occupied frame must be selected for replacement
- Alternatively, the system can promote page replacement as to maintain the list of free frames always with some elements
 - This allows to load the faulty page and free a busy frame at the same time
- The question is: which frame should be selected for replacement?
- An **optimal policy** selects for replacement that page for which the time to the next reference is the longest
 - Unless we have a Crystal-Ball, it is **impossible to implement**
 - But, useful as benchmark
- Covered algorithms:
 - **Least Recently Used (LRU)**
 - **Not Recently Used (NRU)**
 - **First In First Out (FIFO)**
 - **Second chance**
 - **Clock**

Page replacement policies

LRU algorithm

- The **Least Recently Used** policy selects for replacement the frame that has not been referenced the longest
 - Based on the principle of locality of reference, if a frame is not referenced for a long time, it is likely that it will not be referenced in the near future
- Each frame must be labelled with the time of the last reference
 - Additional specific hardware may be required
- On page replacement, the list of occupied frames must be traversed to find out the one with the oldest last access time
- High cost of implementation and not very efficient

Page replacement policies

NRU algorithm

- The **Not Recently Used** policy selects for replacement a frame based on classes
- Bits **Ref** and **Mod**, fields of the entry of the page table, and typically processed by conventional MMU, are used to define classes of frames

class	Ref	Mod
0	0	0
1	0	1
2	1	0
3	1	1

- On page replacement, the algorithm selects at random a frame from the lowest non-empty class
- Periodically, the system traverse the list of occupied frames and put **Ref** at zero

Page replacement policies

FIFO algorithm

- The **FIFO** policy selects for replacement based on the length of stay in memory
 - Based on the assumption that the longer a page resides in memory, the less likely it is to be referenced in the future
- The list of occupied frames is considered to be organized in a FIFO that reflects the loading order of the corresponding pages in main memory
- On page replacement, the frame with the oldest page is selected
- The assumption in itself is extremely fallible
 - Consider for instance system shared libraries
 - But can be interesting with a refinement

Page replacement policies

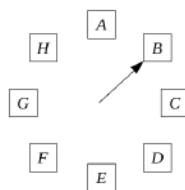
Second chance algorithm

- The **second chance** policy is an improvement of the FIFO algorithm, giving a page a second chance before it is replaced
- On page replacement:
 - The frame with the oldest page is selected as a candidate
 - If its `Ref` bit is at zero, the selection is done
 - If not, the `Ref` bit of the candidate frame is reset, the frame is inserted again in the FIFO, and the process proceeds with the next frame
 - The process ends when a frame with the `Ref` bit at zero is found
- Note that such frame is always found

Page replacement policies

Clock algorithm

- The **clock** policy is an improvement of the second chance algorithm, avoiding the removal and reinsetion of elements in the FIFO
- The list is transformed into a circular one and a pointer signals the oldest element
 - The action of removal followed by a reinsertion corresponds to a pointer advance
- On page replacement:
 - While the `Ref` bit of a frame is non-zero, that bit is reset and the pointer advances to the next frame
 - The first frame with the `Ref` bit at zero is chosen for replacement
 - After replacement, the pointer is placed pointing to the next element



Working set

- Assume that initially only 2 pages of a process are in memory
 - The one containing the first instruction
 - The one containing the start of the stack
- After execution starts and for a while, page faults will be frequent
- Then the process will enter a phase in which page faults will be almost inexistent
 - Corresponds to a period where, accordingly to the principle of locality of reference, the fraction of the address space that the process is currently referencing is all present in main memory
- This set of pages is called the **working set** of the process
- Over time the working set of the process will vary, not only with respect to the number, but also with the specific pages that define it

Thrashing

- Consider that the maximum number of frames assigned to a process is fixed
- If this number is always greater or equal to the number of pages of the different working sets of the process:
 - the process's life will be a succession of periods with frequent page faults with periods almost without them
- If it is lower
 - the process will be continuously generating page faults
 - in such cases, it is said to be in **thrashing**
- Keeping the working set of a process always in memory is a page replacement design challenge

Demand paging vs. prepaging

- When a process transition to the ready state, what pages should be placed in main memory?
- Two possible strategies: **demand paging** and **prepaging**
- **Demand paging** – place none and wait for the page faults
 - inefficient
- **Prepaging** – place those most likely to be referenced
 - first time, the two pages mentioned before (code and stack)
 - next times, those that were in main memory when the process was suspended
 - more efficient

Bibliography

- Operating Systems: Internals and Design Principles, W. Stallings, Prentice-Hall International Editions, 7th Ed, 2012
 - Chapter 7: Memory Management (sections 7.1 to 7.4)
 - Chapter 8: Virtual Memory (sections 8.1 to 8.2)
- Operating Systems Concepts, A. Silberschatz, P. Galvin and G. Gagne, John Wiley & Sons, 9th Ed, 2013
 - Chapter 8: Main Memory (sections 8.1 to 8.5)
 - Chapter 9: Virtual Memory (sections 9.1 to 9.6)
- Modern Operating Systems, A. Tanenbaum and H. Bos, Pearson Education Limited, 4th Ed, 2015
 - Chapter 3: Memory Management (section 3.1 to 3.7)