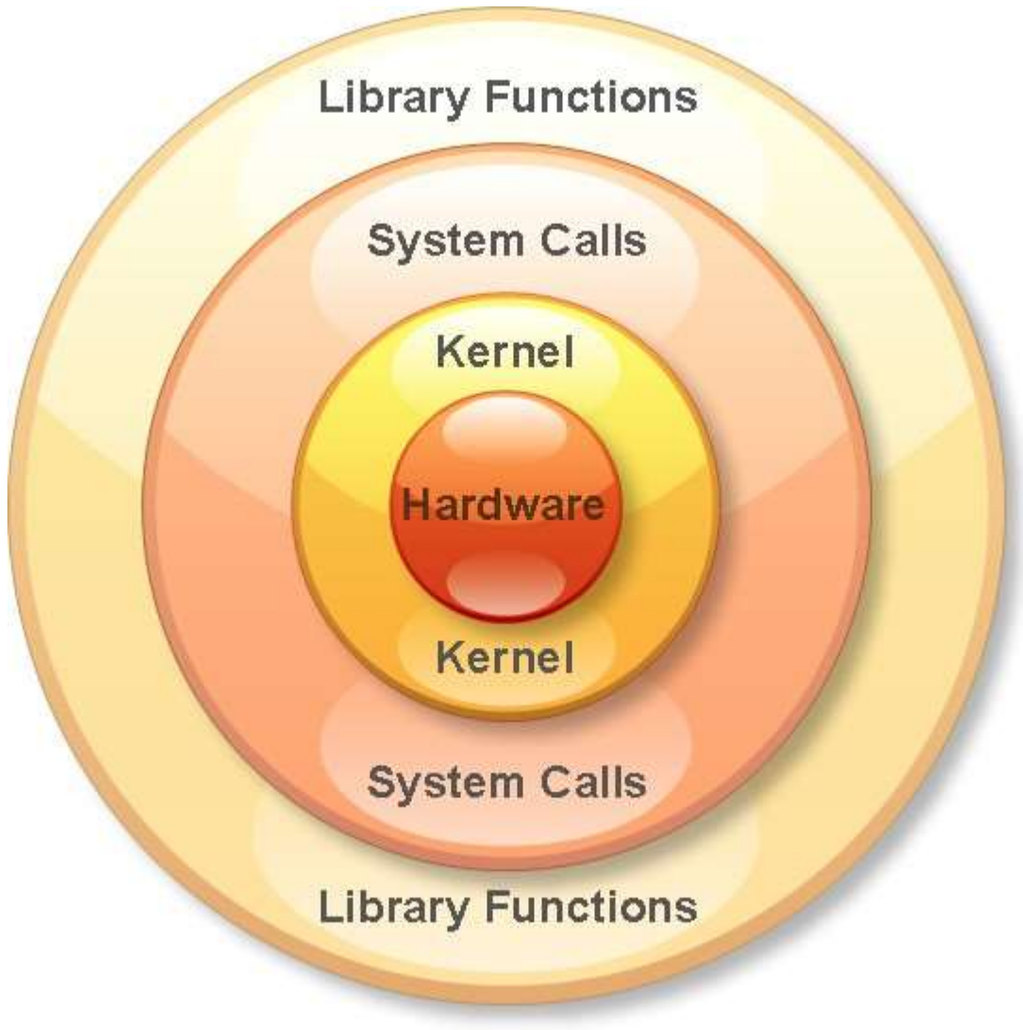


Operating System

Course Objective:

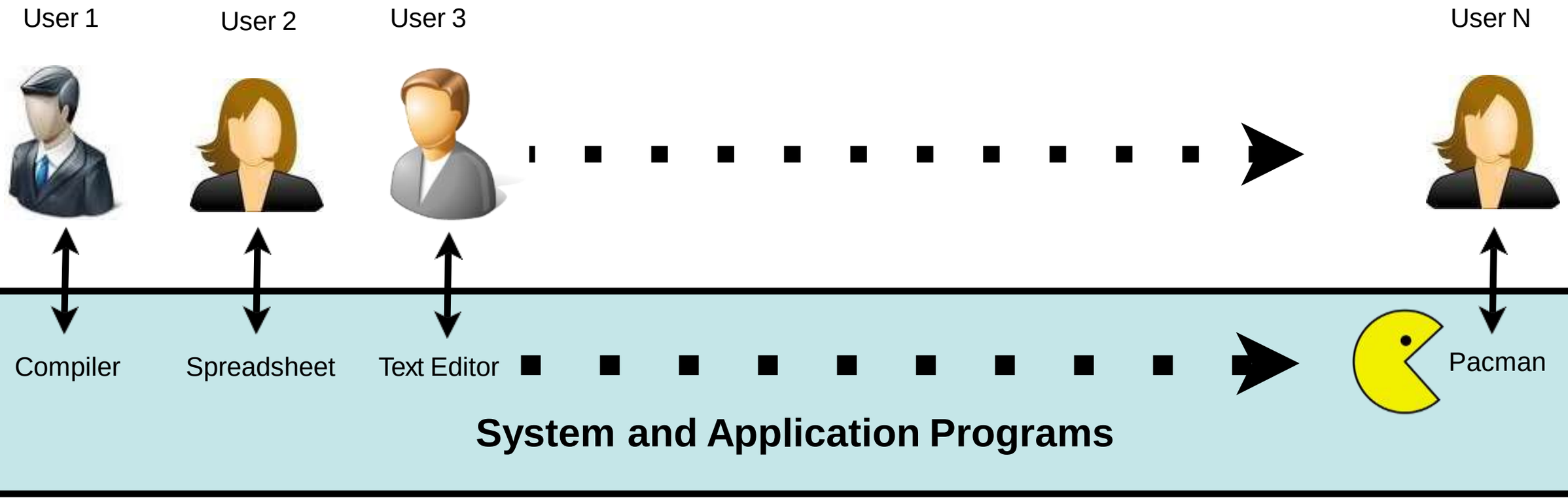
Learn the principles of modern operating systems i.e various functionalities provided by an operating system such as process management, memory management, Storage and I/O management.

Course Outcomes:

1. Analyze the importance and its key principles by differentiating and categorizing the functionalities of an operating system
2. Examine mechanisms involved in memory management to handle processes and threads.
3. Evaluate and solve deadlocks by assessing various handling strategies related to each of the conditions for deadlock.
4. Interpret the mechanisms adopted for file organization and access.
5. Compare and contrast key features and functionality of major operating systems, such as Windows and LINUX.

Learning Resources:

1. Operating System Concepts - Operating System Concepts, Sixth Edition, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, John Wiley & Sons Inc.
2. Modern Operating Systems- Andrew S Tanenbaum, Prentice Hall
3. Operating Systems - Operating System: Internals and Design Principles , William Stallings
4. Operating Systems - System Programming and Operating Systmes D M Dhamdhere, Tata Mc Graw Hill
5. Operating Systems - Operating Systems: A Modern Perspective, Gary Nutt, Addison Wesley
6. Operating Systems - Operating Systems, Achyut S Godbole, Tata Mc Graw Hill
7. Design of the Unix Operating System - Maurice Bach, Prentice Hall.
8. https://nptel.ac.in/courses/106108101/
9. https://www.classcentral.com/course/udacity-introduction-to-operating-systems-3419



Operating System

Controls the hardware and coordinates its use among the various application programs for the various user.

Computer Hardware

What is Operating System?

An operating system is a program that controls the execution of application and acts as an interface between the user of a computer and the computer hardware.

Definition of an operating system can be seen in four aspects:

1) A group of programs that acts as an intermediary between a user and the computer hardware.
2) Controls and co-ordinates the use of computer resources among various application programs and user.
3) Acts as a manager
4) Allow the program to communicate with one another.

Operating System Services

An operating system is an interface between user and hardware and also it provides an ***environment for the execution of programs***.

It ***provides certain services*** to ***programs*** and to the ***users*** of these programs.

These operating system services are ***provided for the convenience of the programmer***, to make the programming task easier.

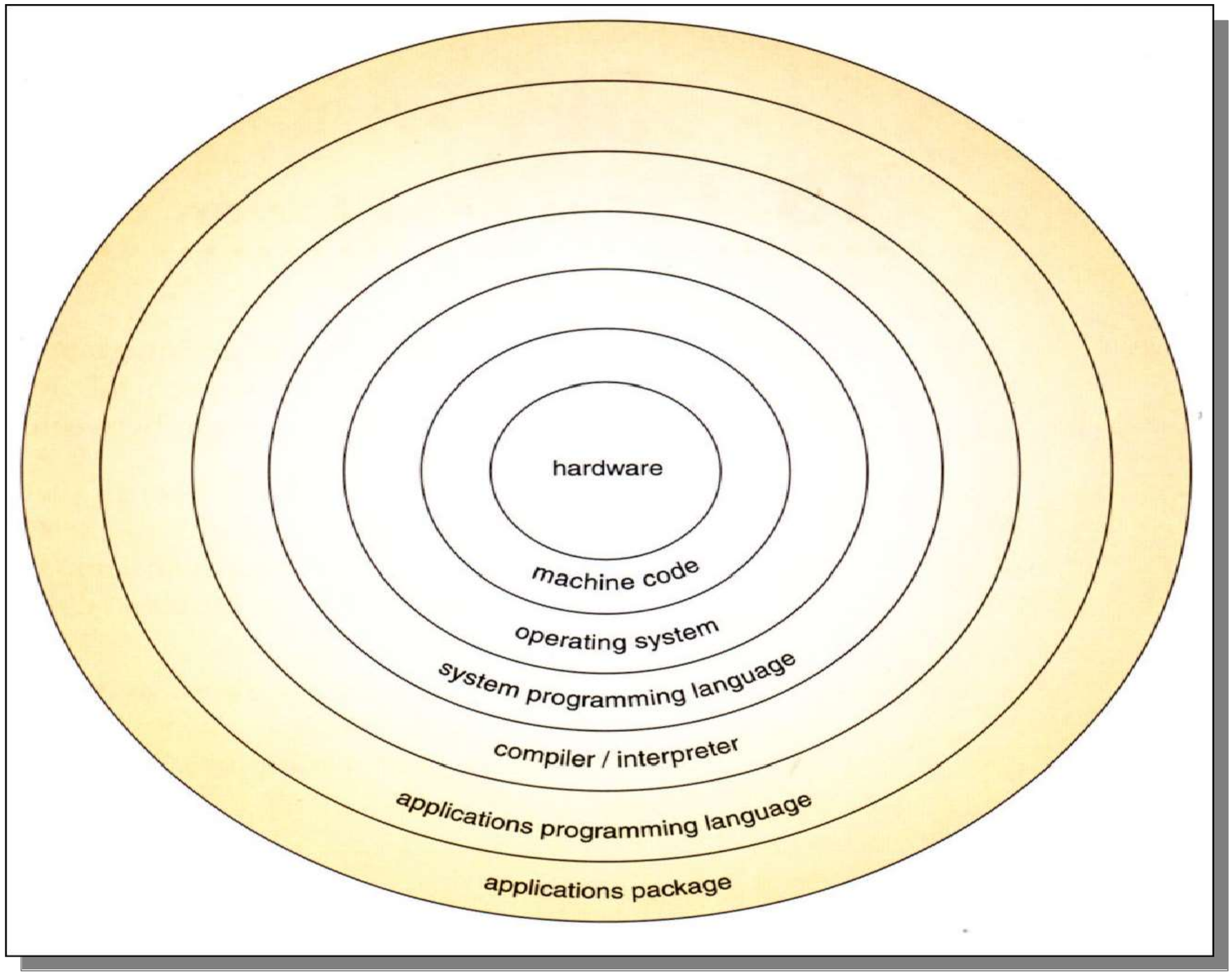
One set of services provides functions that are **helpful to the user**.

Another set of services for ***ensuring the efficient operation of the system itself***.

Operating system Services:

- User Interface.
- Program Execution.
- File system manipulation.
- Input / Output Operations.
- Communication.
- Resource Allocation.
- Error Detection.
- Accounting.

Detail Layered View of Computer



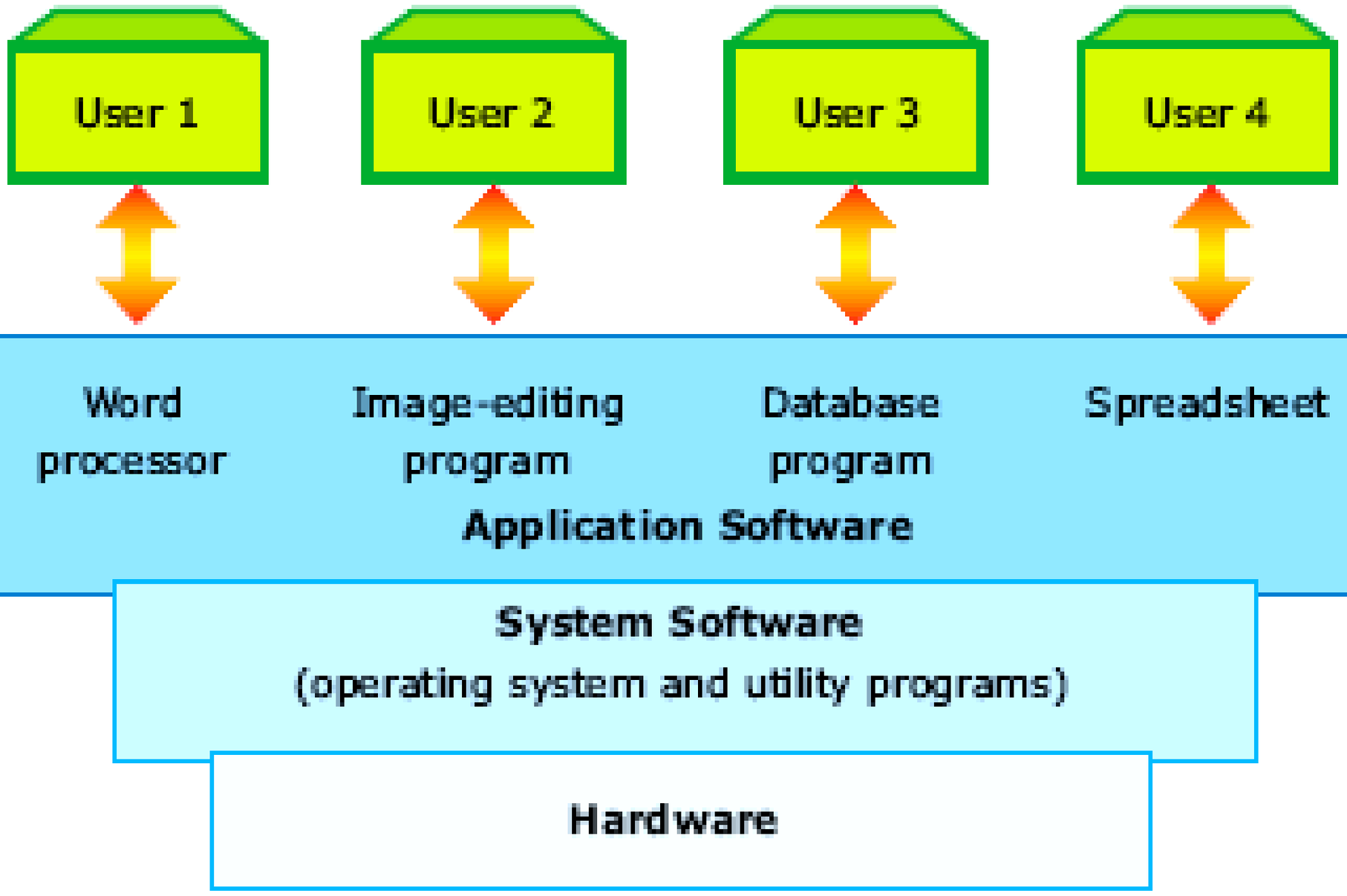
System Software, Application Software and Driver Programs

- System Software- Performs essential operation tasks
  - Operating system
  - Utility programs
- Application Software - Performs specific tasks for users
  - Business application
  - Communications application
  - Multimedia application
  - Entertainment and educational software
- Driver Programs (Device Driver)
  - small program that allows a specific input or output device to communicate with the rest of the computer system

3 types of programs

- user / application programs
  - programs used by the users to perform a task
- system programs
  - an interface between user and computer
- driver programs
  - communicate I/O devices with computer

Hierarchy of computer software



Operating System

- a collection of programs which control the resources of a computer system
- written in low-level languages (i.e. machine-dependent)
- an interface between the users and the hardware
- when the computer is on, OS will first load into the main memory

Basic functions of the operating system

Operating System

Device configuration

Controls peripheral devices connected to the computer

File management

Transfers files between main memory and secondary storage, manages file folders, allocates the secondary storage space, and provides file protection and recovery

Memory management

Allocates the use of random access memory (RAM) to requesting processes

Interface platform

Allows the computer to run other applications

Other function of Operating System

- best use of the computer resources
- provide a background for user's programs to execute
- display and deal with errors when it happens
- control the selection and operation of the peripherals
- act as a communication link between users

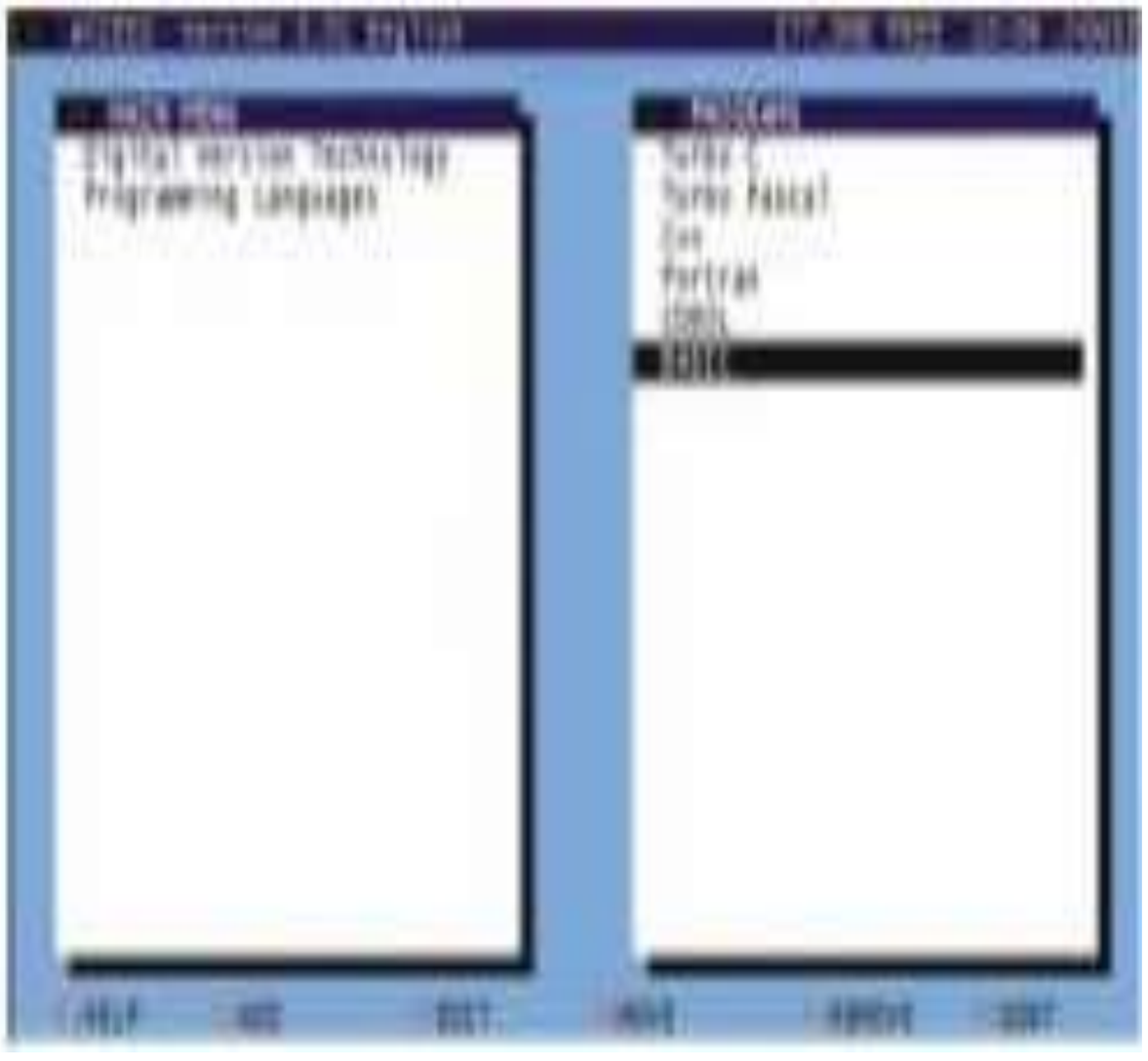
Common Operating Systems and Their Differences

- Network Operating System
  - UNIX / Linux / MS Windows2000 Server
- Desktop Operating System
  - MS Windows 9X/Me / Mac OS / DOS
- Mobile Operating System
  - Palm OS and Pocket PC

Examples

- Common operating systems
  - WINDOW
    - used in IBM compatible microcomputers
  - UNIX
    - multi-user, multi-tasking OS used in minicomputers and microcomputers
  - VAX/VMS
    - used in DEC's VAX series of minicomputers

DOS interface



GUI

Different Types of Operating System

UNIX	DOS	Mac OS	MS Windows	Linux	Palm OS/Pocket PC
Multi-user, multi-tasking	Single-user, single-tasking	Single-user, multi-tasking	Single-user, multi-tasking	Multi-user, multi-tasking	Single-user, multi-tasking
Command-line user interface	Command-line user interface	GUI	GUI	Command-line user interface, GUI	GUI
UNIX has serveral versions but they lack interoperability.	DOS has been replaced by MS Windows OS.	Mac OS has easy-to-use GUI.	The first true MS Windows OS is MS Windows 95.	Linux is an open-source software.	They are specifically designed for PDA.
Network OS	Desktop OS	Desktop OS	Desktop OS	Network OS	Mobile OS

Cross-Platform Issues

- Cross-Platform
  - developing software for, or running software, on more than one type of operating platform.
- Machine-independent Programming Languages
  - JVM
- Makeup Languages
  - HTML
  - XML
- Advantages
  - cost-effective
  - saves time
  - develop the program on different computers

Disk Operating System (DOS)

- a part of operating system to control disk operation
- 2 parts
  - small system data
    - keep track of key information of the disk
  - data area
    - where data file is stored

TYPES OF OS

Distinguished by the response time and how data is entered into the system

- Single user
- Multi user
- Multitasking
- Multi processing
- Embedded
- Real time

SINGLE USER

TWO TYPES:

- Single user, single task
- Single user, multi tasking

Single user, single task

- Designed to manage the computer so that one user can effectively do one thing at a time.
- Example: The Palm OS for Palm handheld computers

Single user, multi tasking

- ➡Designed with a single user in mind but can deal with many applications running at the same time

- ➡ Type of operating system most people use on
- their desktop and laptop computers today

CONT….

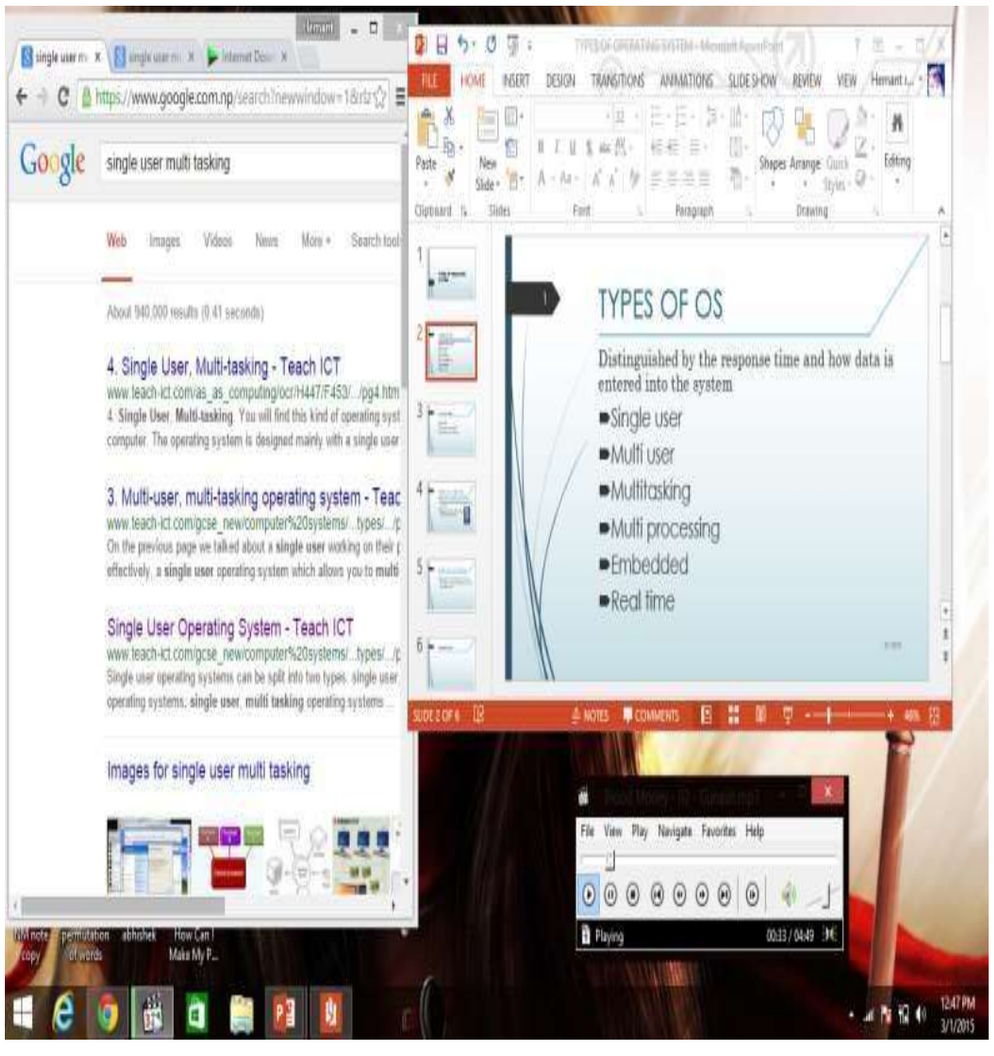
.

- Examples: Microsoft's Windows and Apple's Mac OS platforms

- For Example: It's entirely possible for a Windows user to be writing a note in a word processor while downloading a file from the Internet while printing the text of an e-mail message.

CONT…

.


single user multi tasking
Web
Images
Videos
News
More
Search tools
About 940,000 results (0.41 seconds)
4. Single User, Multi-tasking - Teach ICT
www.teach-ict.com/as_as_computing/ocr/H447/F453/.../pg4.htm
4. Single User, Multi-tasking. You will find this kind of operating syst
computer. The operating system is designed mainly with a single user
3. Multi-user, multi-tasking operating system - Teac
www.teach-ict.com/gcse_new/computer%20systems/...types/.../p
On the previous page we talked about a single user working on their
effectively, a single user operating system which allows you to multi
Single User Operating System - Teach ICT
www.teach-ict.com/gcse_new/computer%20systems/...types/.../p
Single user operating systems can be split into two types: single user
operating systems, single user, multi tasking operating systems ...
Images for single user multi tasking
TYPES OF OS
Distinguished by the response time and how data is entered into the system
Single user
Multi user
Multitasking
Multi processing
Embedded
Real time
SLIDE 2 OF 6
NOTES
COMMENTS
File View Play Navigate Favorites Help
Playing
00:33 / 04:49
12:47 PM
3/1/2015

[2] MULTI USER

- ➡Allows many different users to take advantage of the computer's resources simultaneously
- ➡Allows multiple users to access the computer system at the same time
- ➡Time Sharing system and Internet servers as the multi user systems

CONT

- Examples: UNIX, VMS and Mainfra

[3] MULTI TASKING (or) Time Shared Systems

- Allows more than one program to run concurrently.
- The tasks share common processing resources, such as a CPU and main memory
- In the process, only one CPU is involved, but it switches from one program to another so quickly that it gives the appearance of executing all the programs at the same time.

CONT….

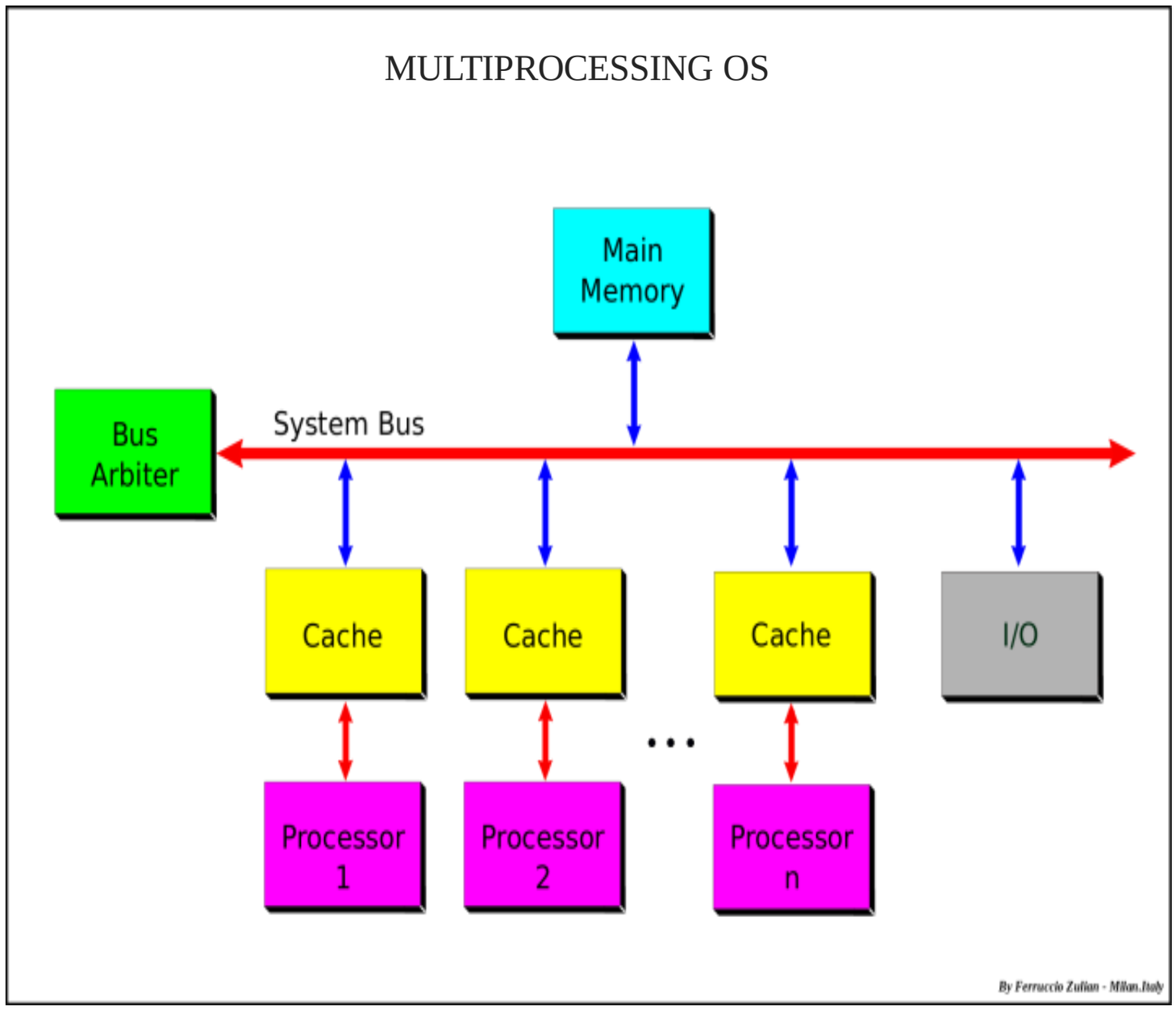
.

screenshot of Debian Linux (version 7.1, "Wheezy") running the GNOME desktop environment, Firefox, Tor, and VLC media player, all at the same time.

[4] MULTI PROCESSING

- ➡Multiprocessing, in general, refers to the utilization of multiple CPUs in a single computer system
  - ➡Enables several programs to run concurrently
    - ➡The term also refers to the ability of a system to support more than one processor and/or the ability to allocate tasks between them

CONT


MULTIPROCESSING OS
Main Memory
Bus Arbiter
System Bus
Cache
Cache
Cache
I/O
Processor 1
Processor 2
Processor n
By Ferruccio Zulian - Milan,Italy

[5] EMBEDDED OS

[4][5]

- Designed to be used in embedded computer systems
- Are able to operate with a limited number of resources on small machines like PDAs
- Are very compact and extremely efficient by design
- is a computer that is part of a different kind of machine
- Examples include computers in cars, digital televisions, ATMs, airplane controls, digital cameras, GPS navigation systems, elevators, and among many other possibilities.

CONT…

…

Embedded OS in a car

Android OS in digital camera

[6] REAL TIME OPERATING SYSTEM

- is a multitasking operating system that aims at executing real-time applications
- The main objective of real-time operating systems is their quick and predictable response to events
- In it, the time interval required to process and respond to inputs is so small that it controls the environment

CONT…

.

- Examples: QNX, RTLINUX
- Are used to control machinery, scientific instruments and industrial systems

CONT….

.


virtualization and real-time systems
AEROSPACE & DEFENSE
AVIONICS
MEDICAL
SECURE CLIENT
NETWORK SECURITY
EMBEDDED DEVICES

Components of OS

- Process Management
- Memory management
- I/O Device management
- File system
- Protection
- Network management
- Network services
- User Interface

Examples of computing devices which use OS

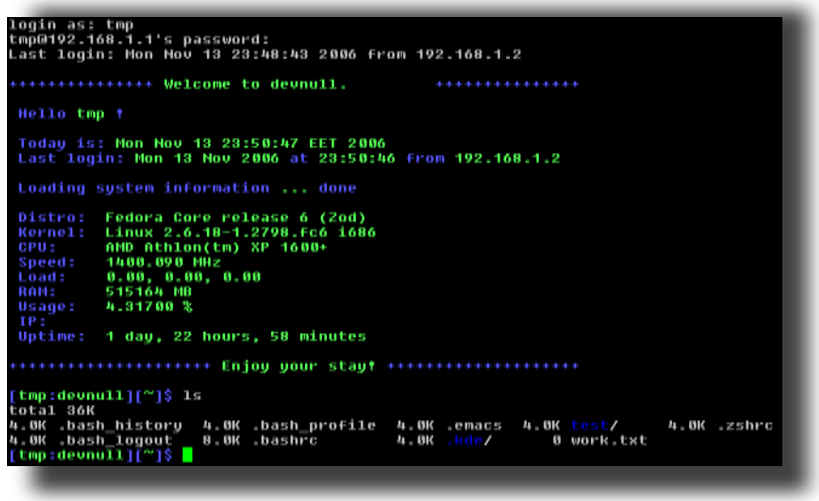
- Computers
- Mobile phones
- 3d televisions
- Video game
- ATM
- Ticket Wending Machine

User Interface

Types of User Interface

Graphical User Interface (GUI)

Usually a window system with a pointing device to direct I/O, choose from menus and make selections and a keyboard to enter text.



Command Line Interface (CLI)

A mechanism for interacting with a computer operating system or software by typing commands to perform specific tasks.

Batch Interface (shell scripting)

Commands and directives to control those commands are entered into files, and those files are executed.

Many systems now include both CLI and GUI interfaces

Apple Mac OS X has “Aqua” GUI interface with UNIX kernel underneath and shells available.

Microsoft Windows is GUI with CLI “command” shell.

Solaris is CLI with optional GUI interfaces (Java Desktop, KDE).

Command Interpreter

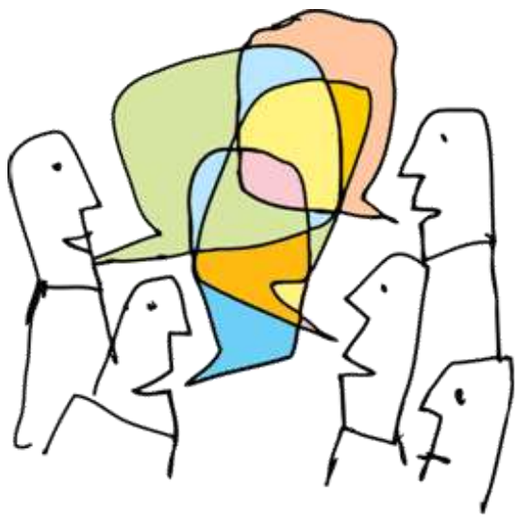
The command interpreter allows users to enter commands directly to be performed by the operating system.

★ Some operating systems(Vx works) include the command interpreter in the kernel.

★ Others, such as Windows and Linux, treat the command interpreter as a special program that is running when a job is initiated or when the user firs logs on.

On systems with multiple command interpreters to choose from, the interpreters are know as ***shells***.

The main function of the command interpreter is to get and execute the next user-specified command.



Command Interpreter

Let’s look at an example using the Linux approach.

Example: Use the command line to delete the file file.txt

```
$> rm file.txt
```

1) read command line ”rm file.txt” from the user.
2) parse the command line (split into command ”**rm**” and parameter ”**file.txt**”.
3) search for a file called **rm**.
4) execute the **rm** program with the parameter **file.txt**.

System and Application Programs

GUI

batch

command line

user interfaces

system calls

program execution

I/O operations

communication

Helpful for the user

error detection

file systems

resource allocation

accounting

ensuring the efficient operation of the system itself

protection and security

Services

Operating System

Computer Hardware

Services useful for the user

program execution

The system must be able to ***load*** a program into memory and to ***run*** that program. The program must be able to ***end*** its execution, either normally or abnormally (indicating error).

I/O operations

Users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

file systems

Programs needs to ***read*** and ***write*** files and directories. They also need to ***create*** and ***delete*** them by name, ***search*** for a given file and list file information. May want to ***restrict access*** to files or directories based on ownership.

communication

Processes needs to exchange information. Communication can be implemented via ***shared memory*** or through ***message passing***.

error detection

Errors may occur in the CPU and memory hardware, in I/O devices (network failure, out of paper, etc...) and in user programs (arithmetic overflow, attempts to access an illegal memory location).

For each type of error, the operating system should take the appropriate action to ensure correct an consistent computing.

Services for ensuring the efficient operation of the system itself.

resource allocation

When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Critical resources: ***CPU time, main memory and file storage***.

accounting

May want to keep track of which users use how much and what kind of resources. Could be useful for ***billing*** or for ***usage statistics***.

protection and security

When several separate processes execute concurrently, ***it should not be possible for one process to interfere with the others or with the operating system itself***. Security of the system from outsiders is also important.

Exceptions and interrupts

Interrupts and exceptions are used to notify the CPU of events that needs immediate attention during program execution.

Exceptions are internal and synchronous

★ Exceptions are used to handle internal program errors.

★ Overflow, division by zero and bad data address are examples of internal errors in a program.

★ Another name for exception is trap. A trap (or exception) is a software generated interrupt.

★ Exceptions are produced by the CPU control unit while executing instructions and are considered to be synchronous because the control unit issues them only after terminating the execution

Interrupts are external and asynchronous

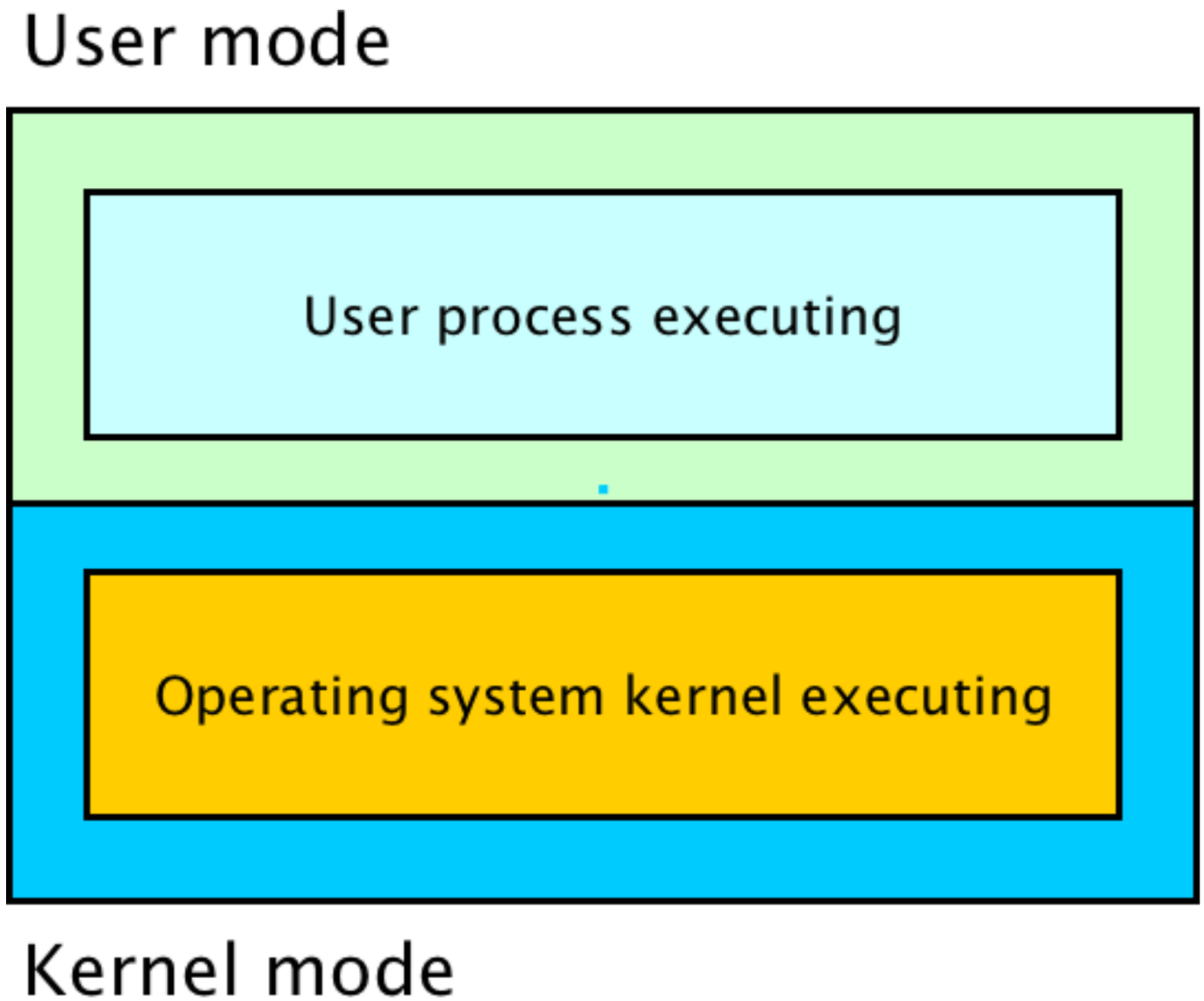
- ★ Interrupts are used to notify the CPU of external events.
- ★ Interrupts are generated by hardware devices outside the CPU at arbitrary times with respect to the CPU clock signals and are therefore considered to be asynchronous.
- ★ Key-presses on a keyboard might happen at any time.
- ★ Read and write requests to disk is similar to key presses.

Exception and interrupt handling

- ★ When an exception or interrupt occurs, execution **transition** from user mode to kernel mode.
- ★ The **cause** of the interrupt or exception is determined.
- ★ The exception or interrupt is **handled**.
- ★ When the exception or interrupt has been handled execution **resumes** in user space.

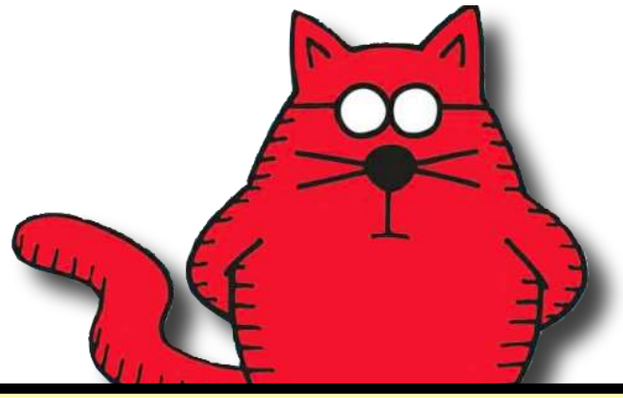
Dual mode operation

In order to protect the operating system from user processes and protect user processes from each other, two modes are provided by the hardware: user mode and kernel mode.



Dual mode operation place restrictions on the type and scope of operations that can be executed by the CPU. This design allows the operating system kernel to execute with more privileges than user application processes.

Dual Mode of Operation



Modern operating systems are event driven. Events are almost always signaled by the occurrence of an **interrupt** or a **trap**.

At system boot time, the hardware starts in monitor mode.

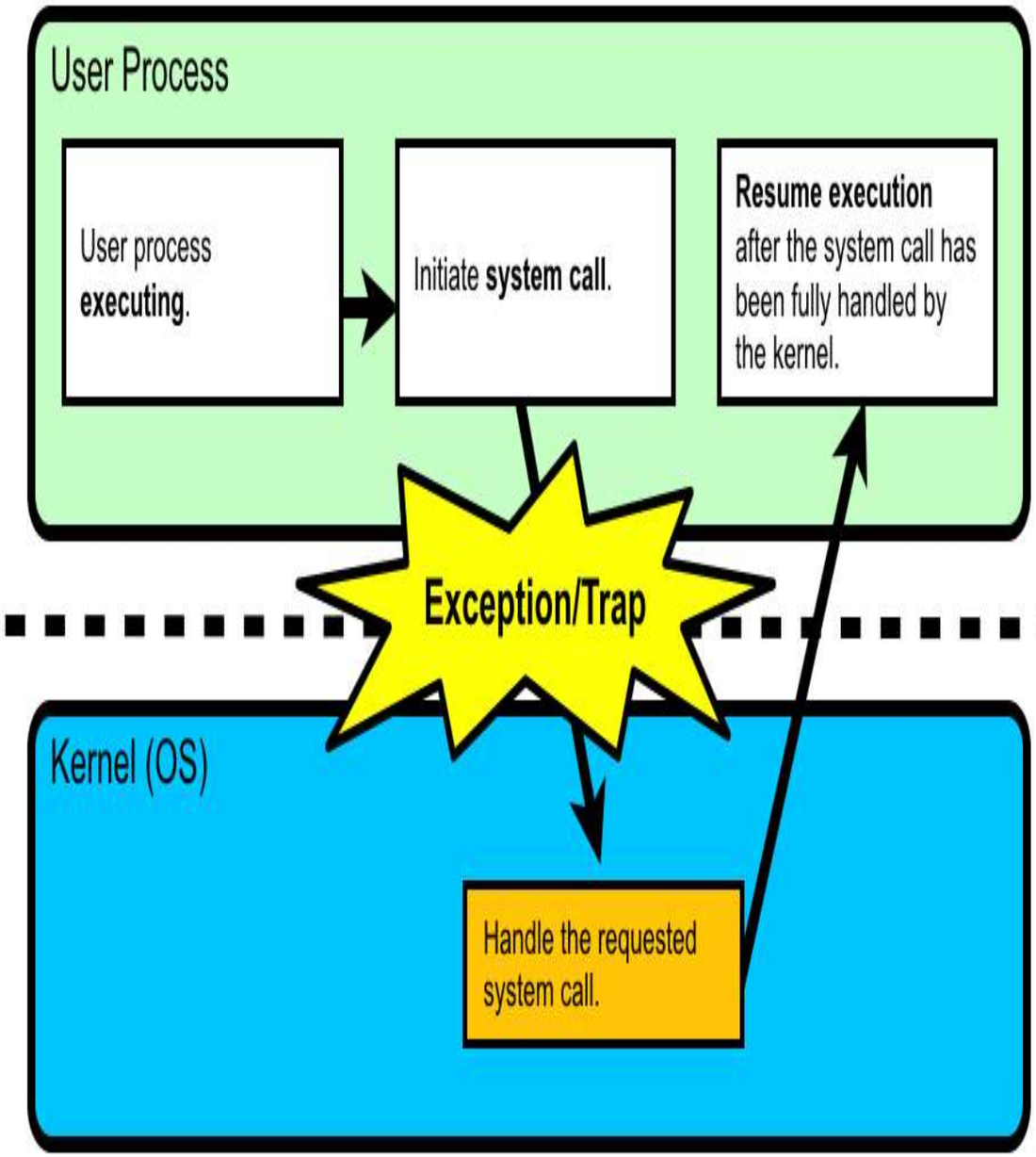
Whenever a trap or interrupt occurs, the hardware switches from user mode to monitor mode

changes the state of the mode bit to be 0

system always **switches to user mode** after Servicing the user request

by setting the mode bit to 1

before passing control to a user program.



Mode bit=1

User Mode

Kernel Mode

Mode bit=0

System and Application Programs

GUI

batch

command line

user interfaces

system calls

program execution

I/O operations

communication

Helpful for the user

error detection

file systems

resource allocation

accounting

ensuring the efficient operation of The system itself

protection and security

Services

Operating System

Computer Hardware

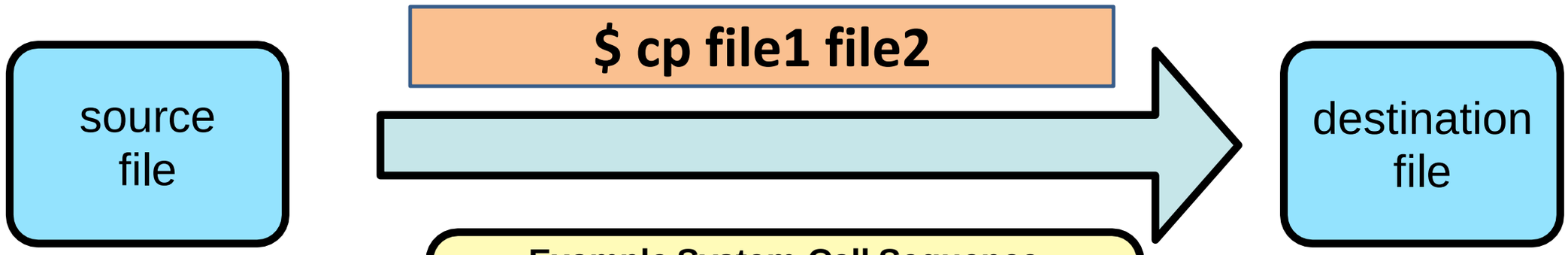
System Calls

System calls provide an interface to the services made available by an operating system.

★ These calls are generally available as routines written in C and C++.

★ Certain low level tasks (for example, tasks where hardware must be accessed directly)

Example: a simple program to read data from one file and copy them to another file



Even simple programs may make heavy use of the operating system.

Example System Call Sequence

Acquire input file name:
    Write prompt to screen
    Accept **input**

Acquire output file name:
    Write prompt to screen
    Accept **input**

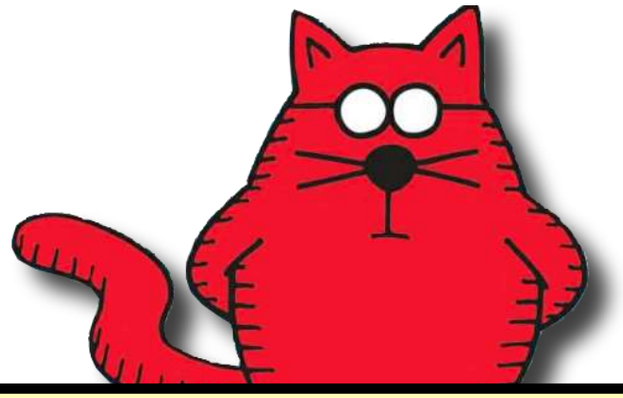
Open the input file:
    If file doesn't exits, abort

Create output file:
    if file exists, abort

Loop:
    Read from input file
    Write to output file
Until read fails

Close input file
Close output file
Write completion message to screen
Terminate normally

Frequently, systems execute thousands of system calls per second.



Most programmers never sees this level of detail.

Typically, application developers design programs according to an ***application programming interface*** (API).

Example System Call Sequence

Acquire input file name:
    Write prompt to screen
    Accept **input**

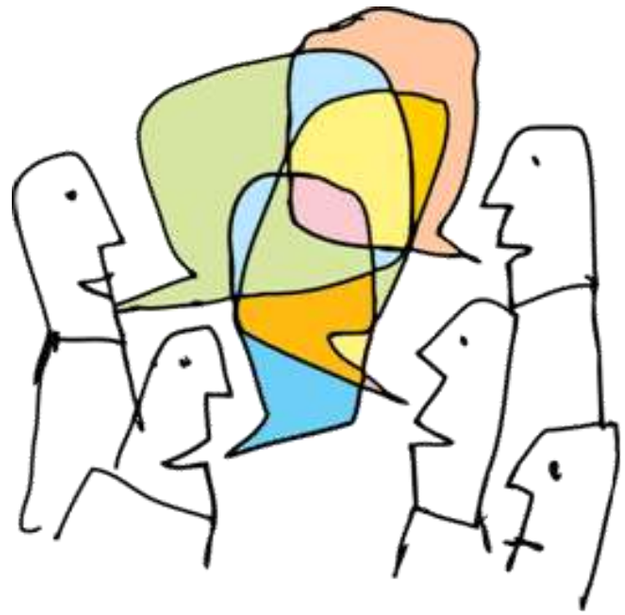
Acquire output file name:
    Write prompt to screen
    Accept **input**

Open the input file:
    If file doesn't exits, abort

Create output file:
    if file exists, abort

Loop:
    Read from input file
    Write to output file
Until read fails

Close input file
Close output file
Write completion message to screen
Terminate normally



Why would an application programmer prefer programming according to an API rather than invoking actual system calls?

Application Programming Interfaces

The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.

Three of the most common APIs:

Win32 API for Windows systems.

The **P**ortable **O**perating **S**ystem **I**nterface for Uni**x (POSIX) API** for Unix like systems.

The **Java API** for designing programs that run on the Java virtual machine.

Win32 API

POSIX API

Java API

One benefit of programming according to an API concerns ***program portability***.

Further more, ***actual system calls can after be more detailed*** and difficult to work with than the API available to an application programmer.

NOTE: There often exists a strong correlation between a function in the API and its associated system call within the kernel. In fact, many of the POSIX and Win32 APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

User Process

```
open();   // invoking the open() system call
```

GUI

batch

command line

user interfaces

system call Interface

program execution

I/O operations

communicati
on

Helpful for the user

error detection

file systems

resource allocation

accounting

ensuring the efficient operation of the system itself

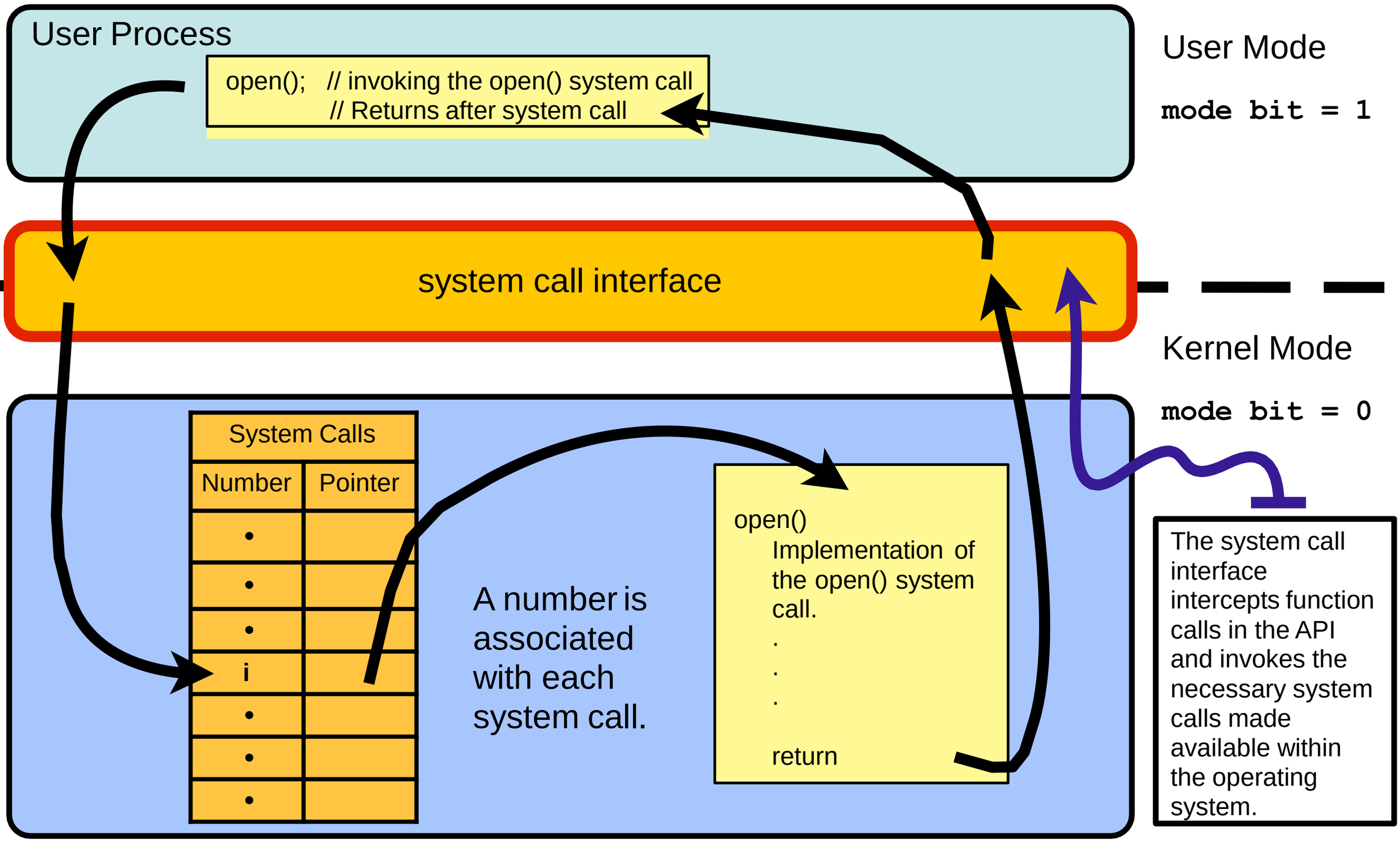
protection and
security

Services

Operating System

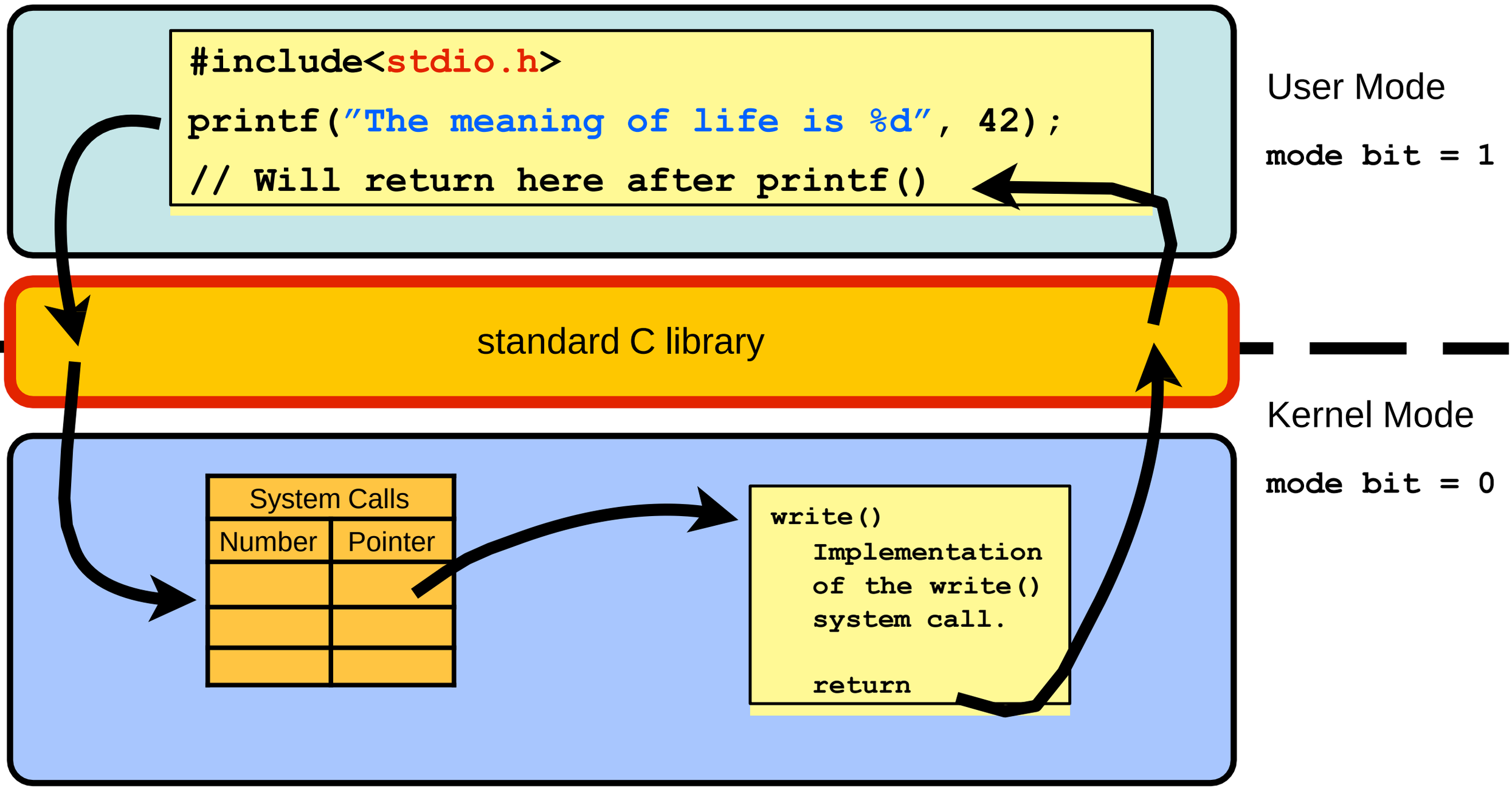
Computer Hardware

API – System Call – OS Relationship



The C Standard Library

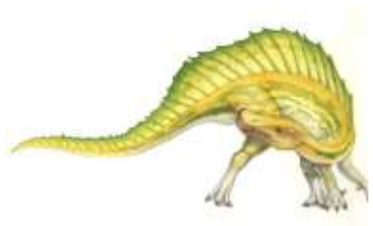
The C standard library provides a portion of the system-call interface for many versions of UNIX and Linux.



In this example, the printf() statement is intercepted by the standard C library. The standard C library takes care of constructing the string to print and invokes the write system call to actually print the string.

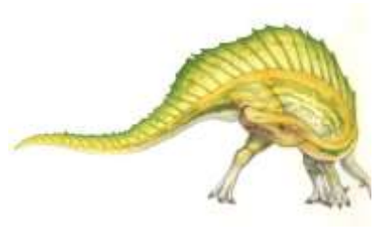
Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event----------concurrent process
  - allocate and free memory
  - Dump memory if error-------debugging.
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes



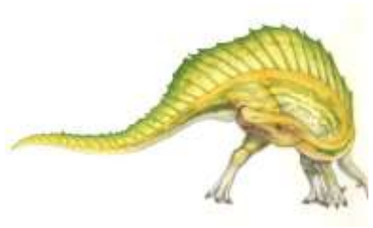
Types of System Calls

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices



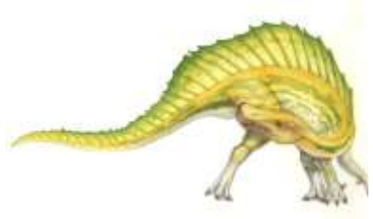
Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices



Types of System Calls (Cont.)

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access



Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

The process concept and inter process communication

Input devices

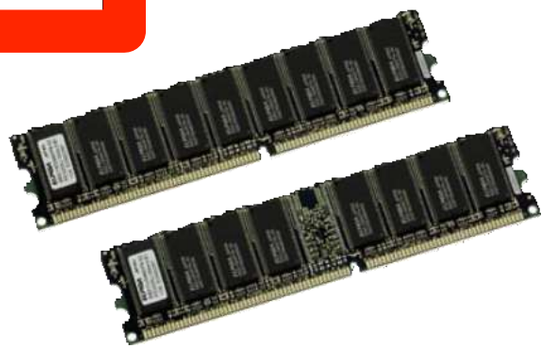
CPU

Control unit

Arithmetic logic unit

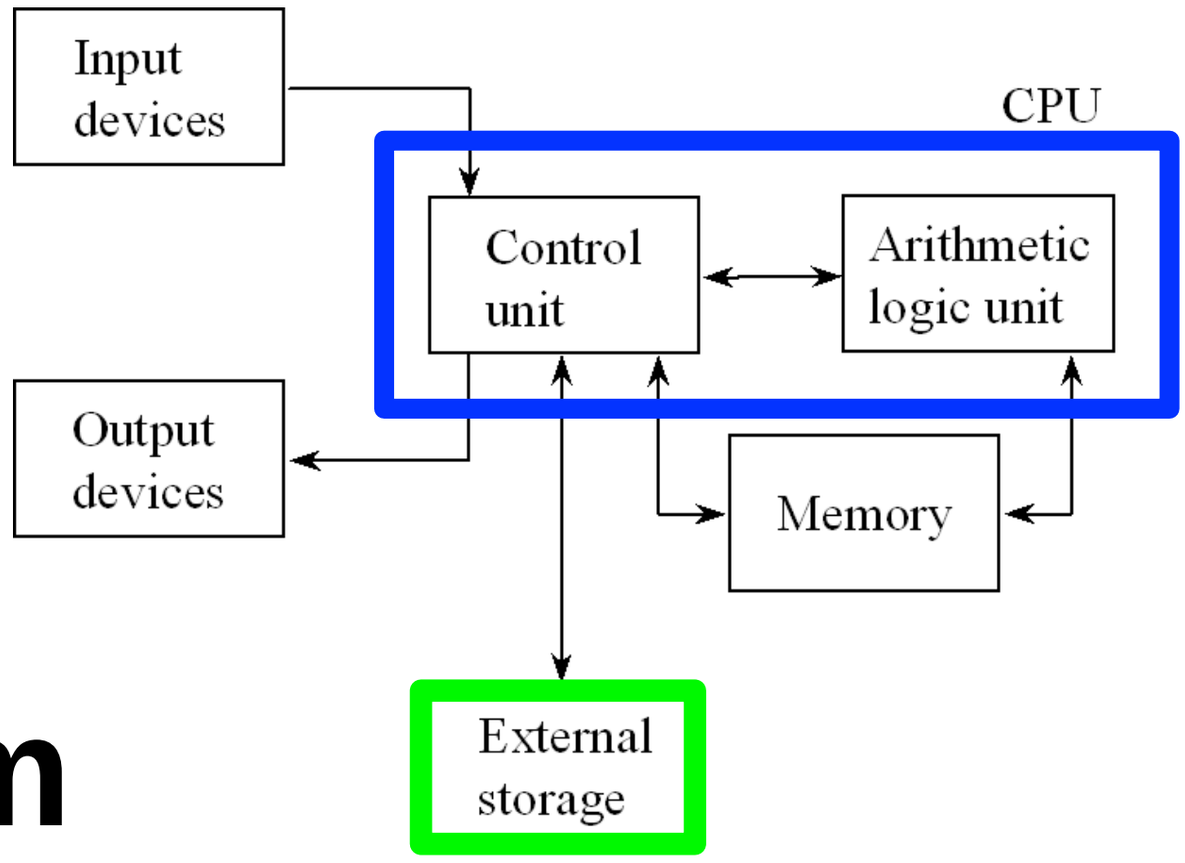
Output devices

Memory



External storage

The von Neuman modell

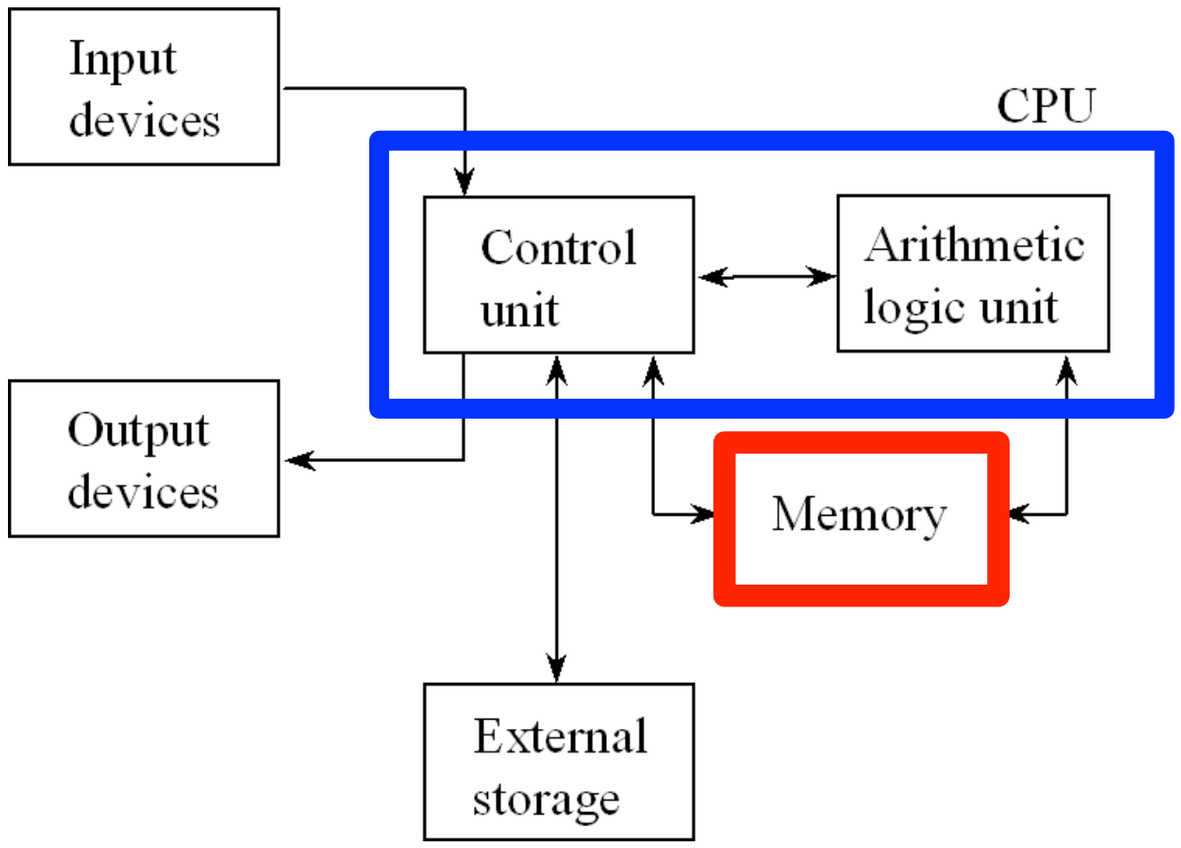


Program

A set of instructions which is in human readable format. A passive entity stored on secondary storage (external storage).

Executable

A compiled form of a program including machine instructions and static data that a computer can load and execute. A passive entity stored on secondary storage (external storage).

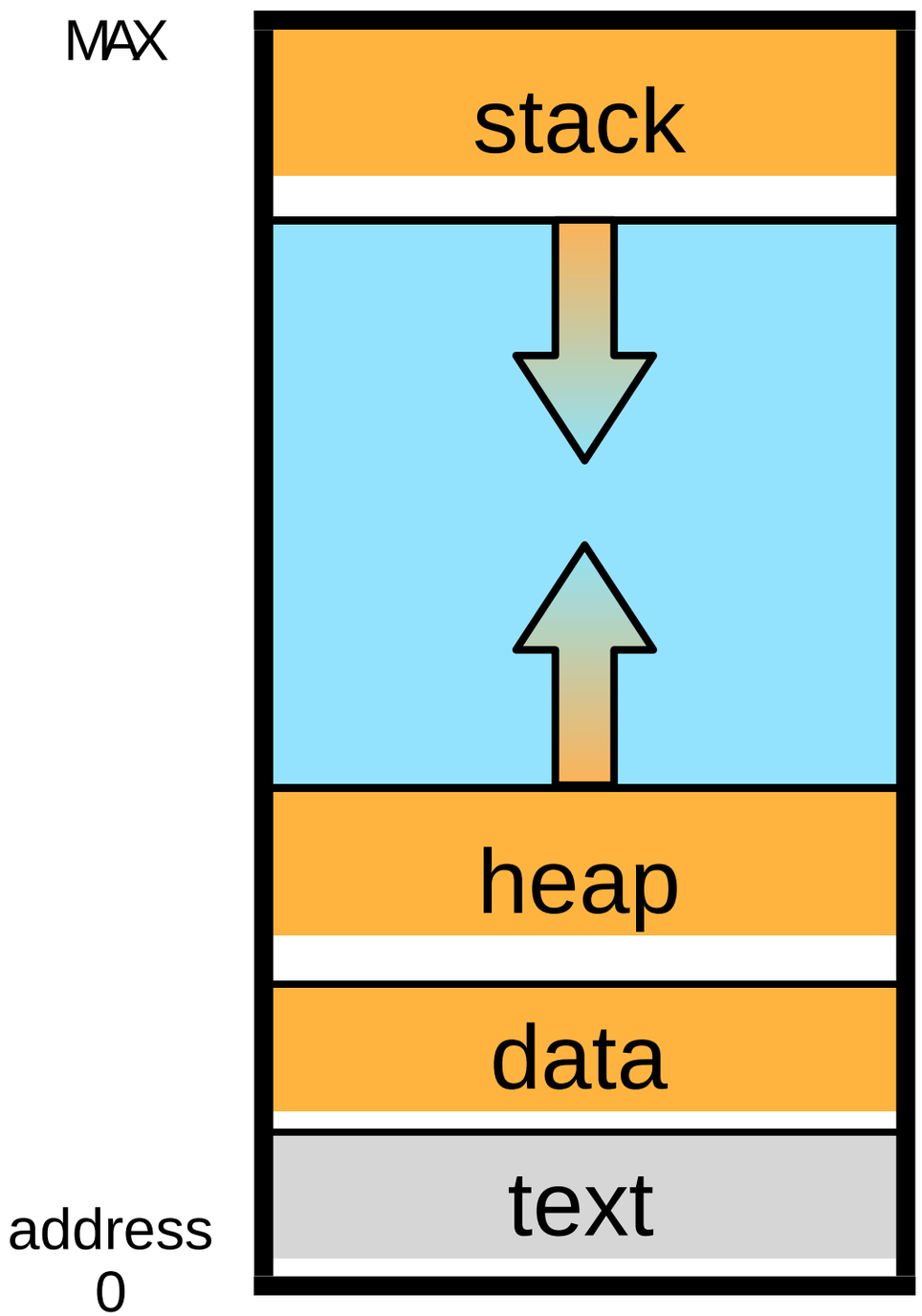


Process

A **program** loaded into **memory** and executing or waiting. A process typically executes for only a short time before it either finishes or needs to perform I/O (waiting).

A process is an active entity and **needs resources** such as **CPU time**, **memory** etc to execute.

A process in memory



Stack: Temporary data such as function parameters, local variables and return addresses.

The stack grows from high addresses towards lower address.

Heap: Dynamically allocated (malloc) by the program during runtime.

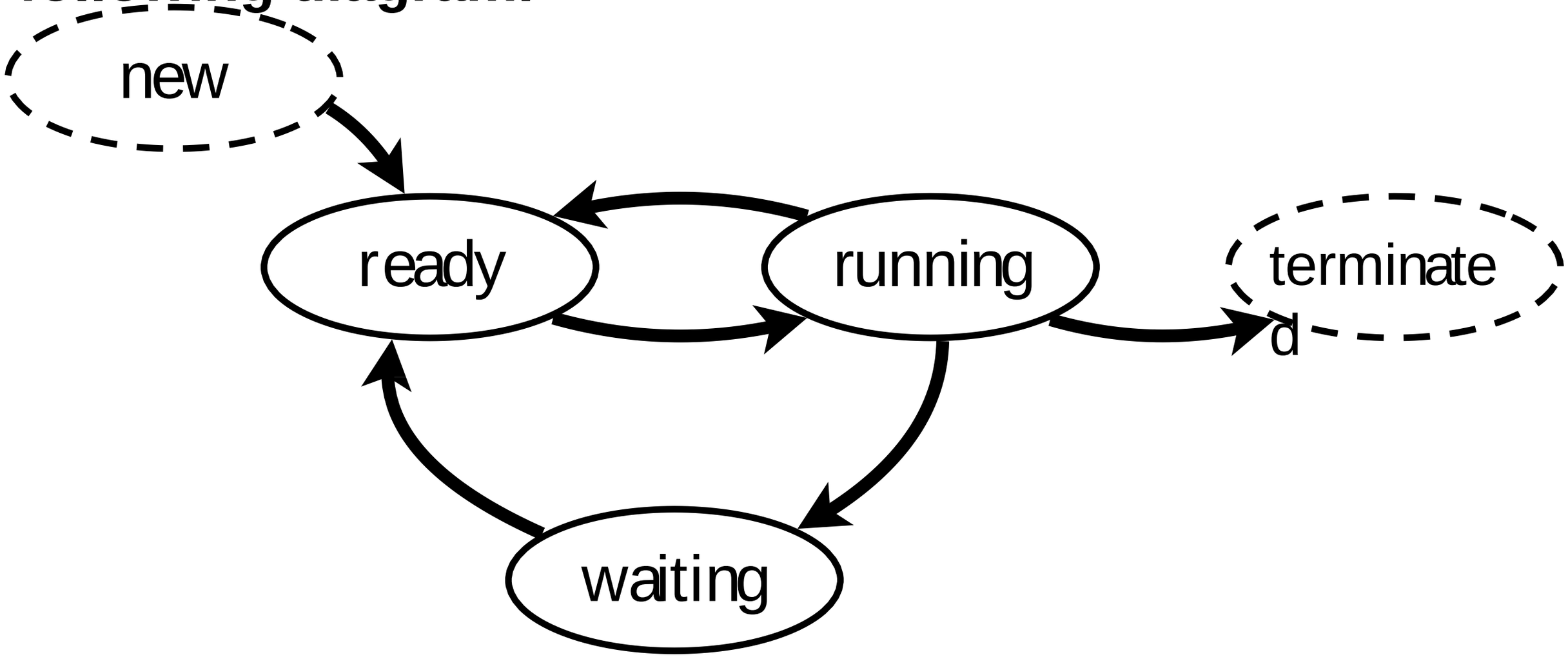
The heap grows from low addresses towards higher addresses.

Data: Statically (known at compile time) global variables and data structures.

Text: The program executable machine instructions.

Process state transitions

In general, a process can be in one of five states: new, ready, running, waiting or terminated. A process transitions between the states according to the following diagram.



Exampleprogram

example.c

```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
   int y;
   char* str;
   str = malloc(50);
   exit(EXIT_SUCCESS);
}
```

Compiler

The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

Source https://en.wikipedia.org/wiki/Compiler

Executable

A compiler (for example gcc) transforms a program to an executable.

```
$ gcc example.c
```

In this example the compiler **gcc** transformed the **example.c** program into the file **a.out**.

```
$ ls
a.out
example.c
```

Process

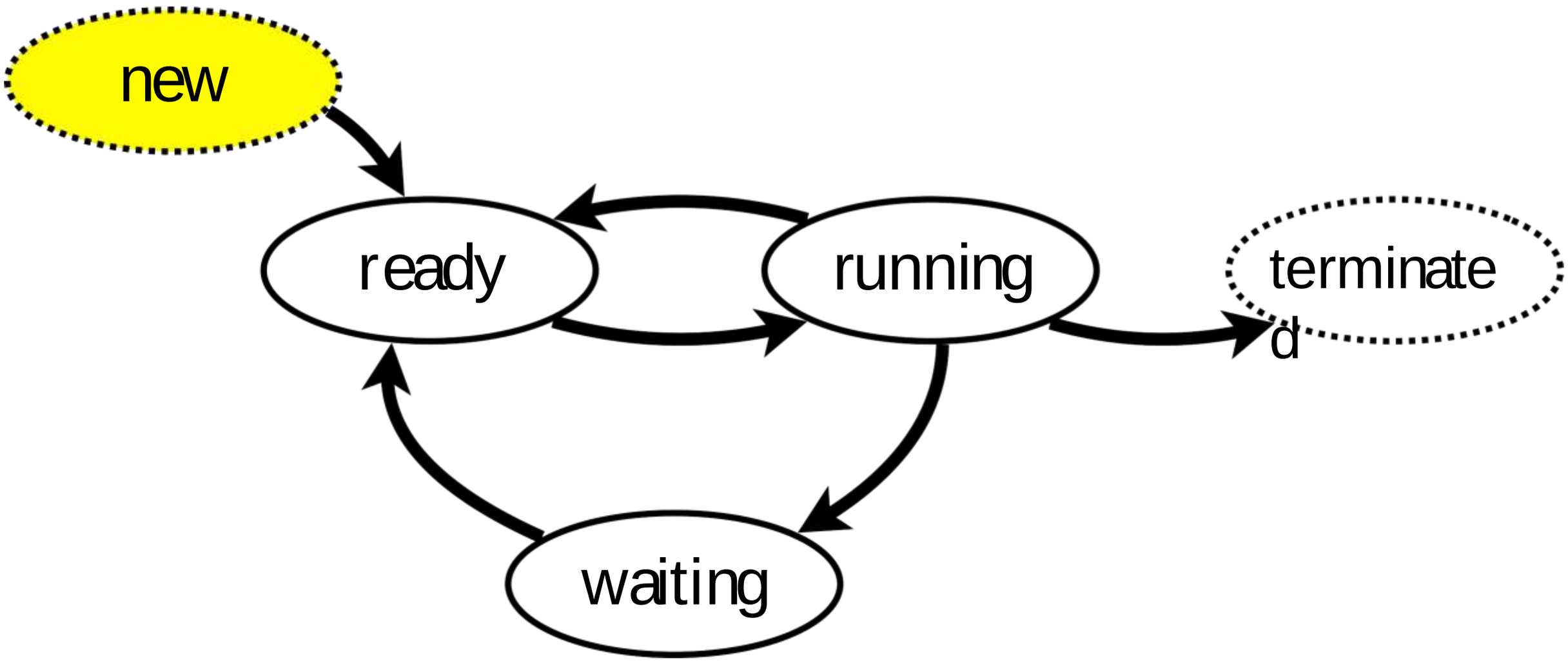
To run the program, the operating system must first create a process.

```
$ ./a.out
```

To create a process, the operating system must allocate memory for the process.

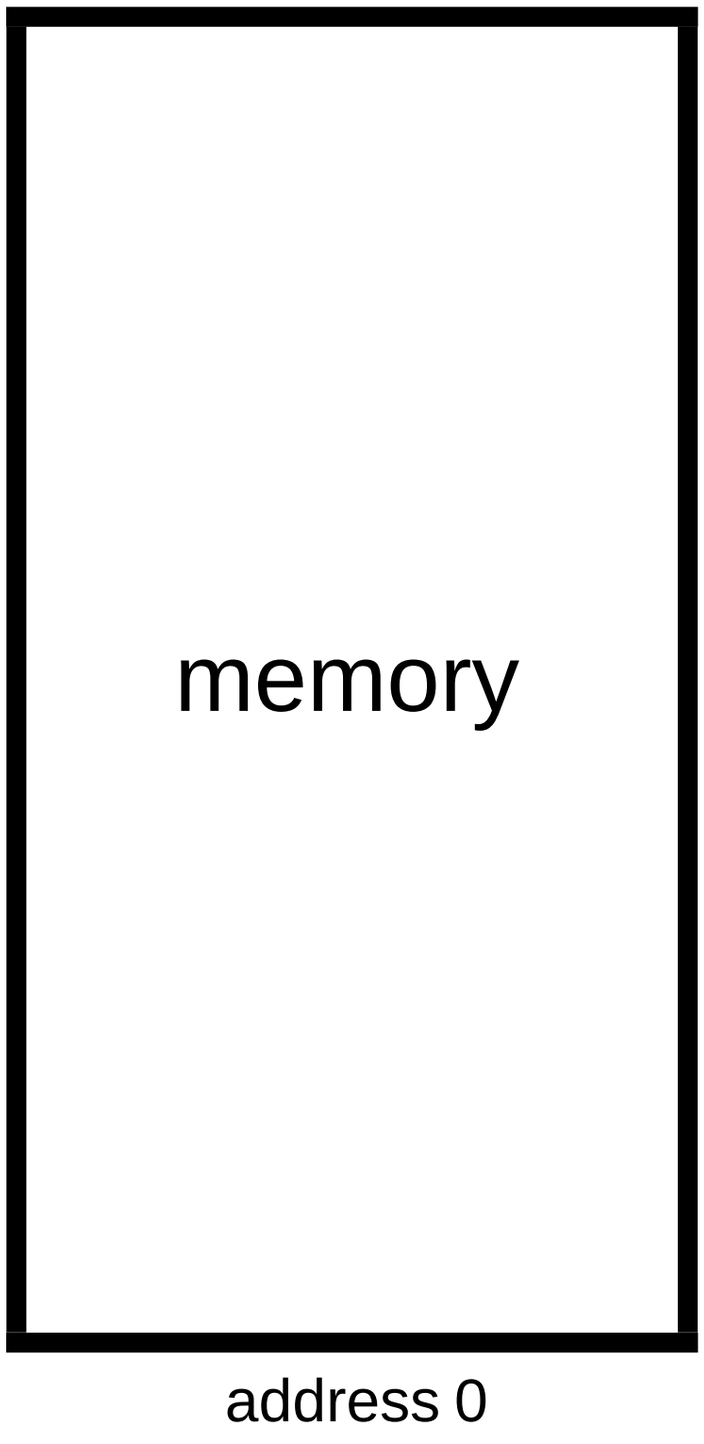
New

To run a program, the operating system must first allocate memory for the process.



Static memory allocation

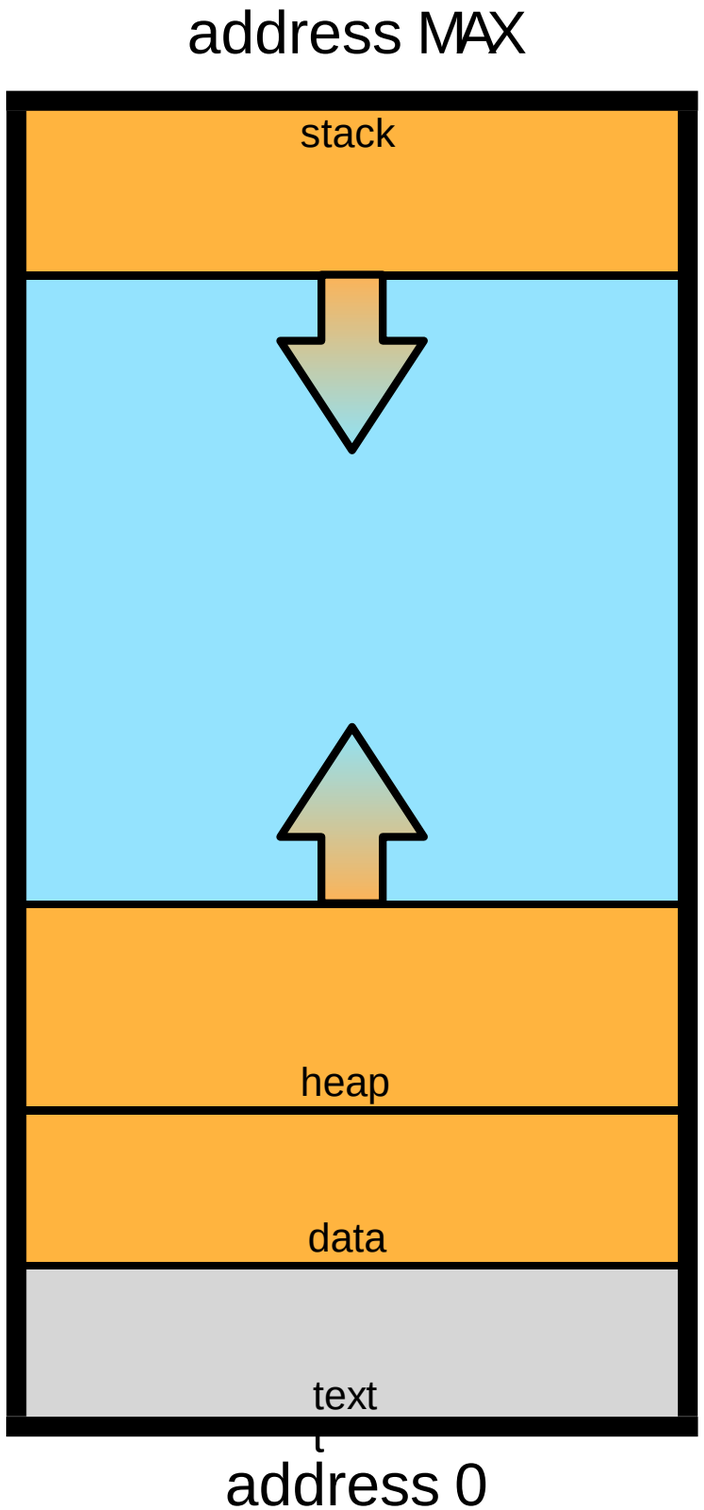
```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```



The operating system allocates a blob of memory for the new process.

Static memory allocation

```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```

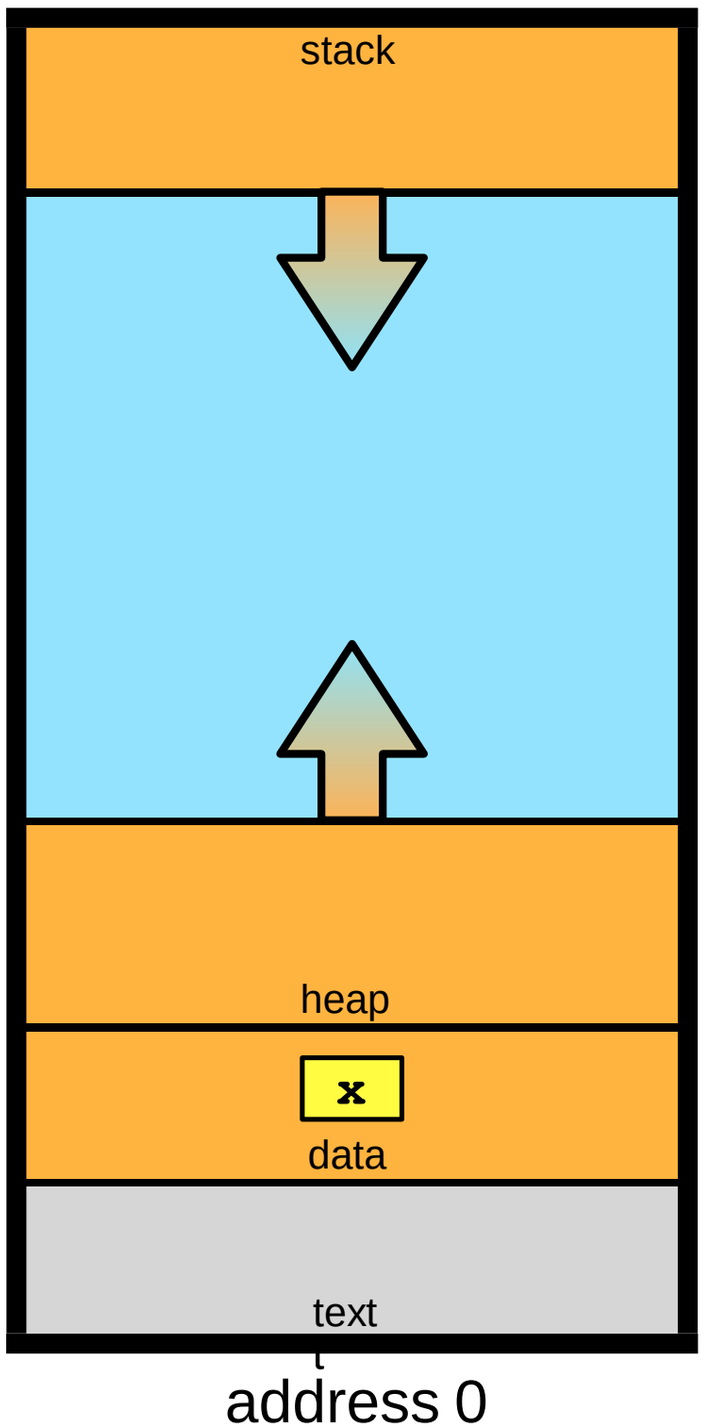


The allocated memory is divided into the following segments:

- text
- data
- heap
- stack

Static memory allocation

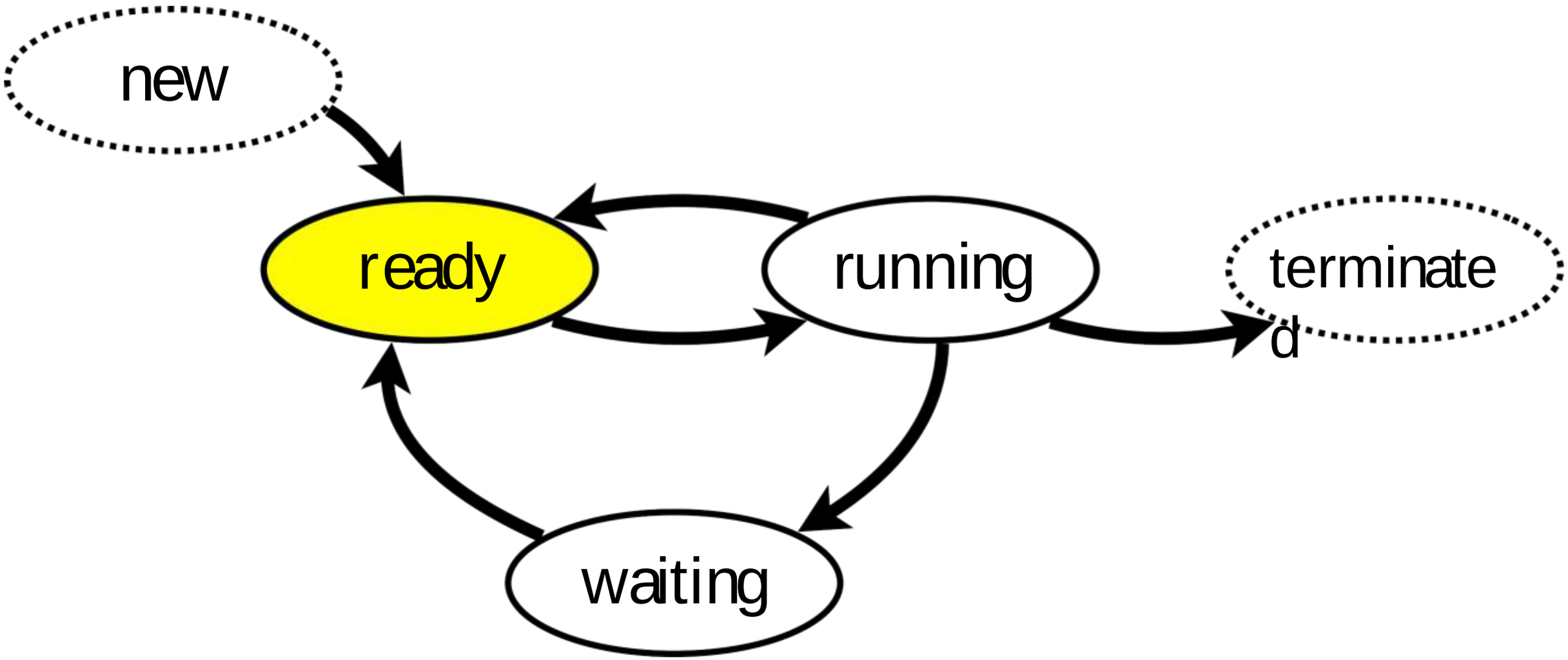
```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```



The size of the needed storage for the **global variable `x`** is known at compile time and storage for x is allocated in the static **data segment**.

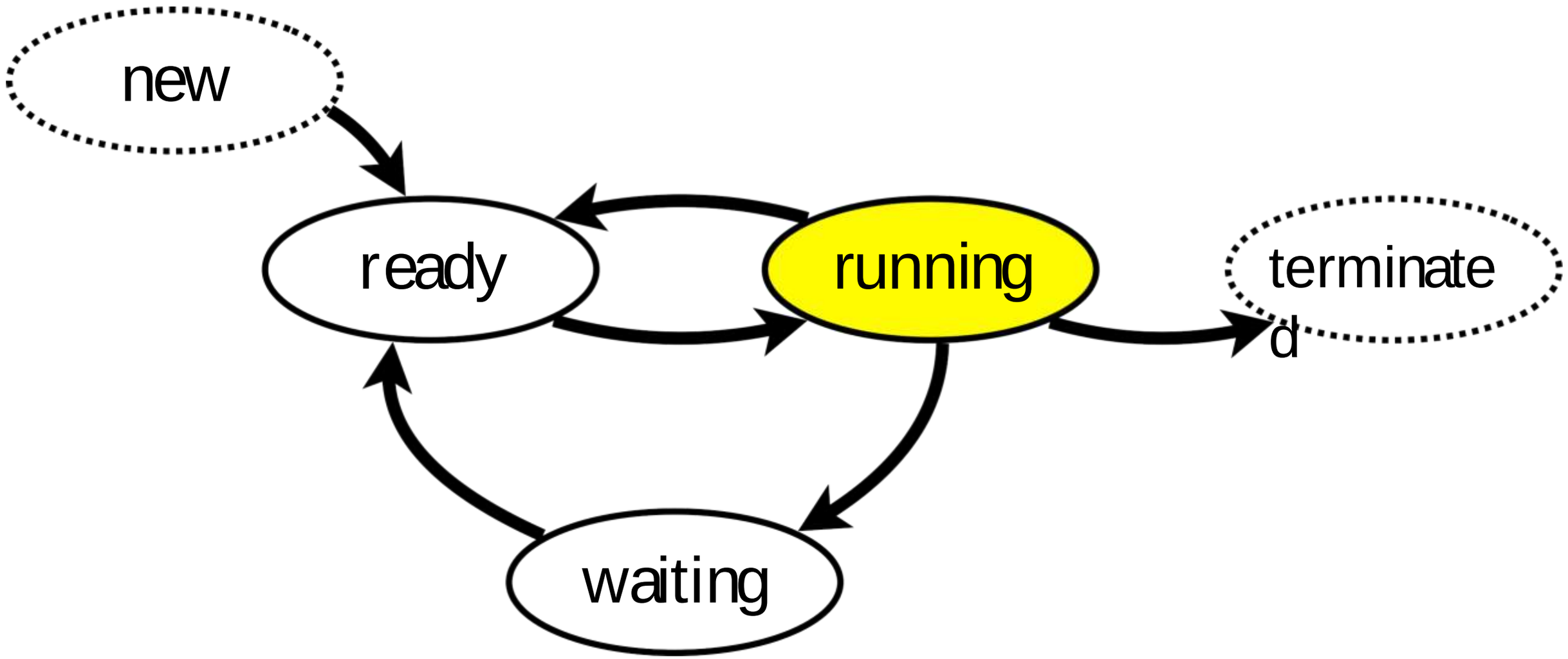
Ready

After the operating system has allocated memory for the process it is ready to execute and changes state from new to ready.



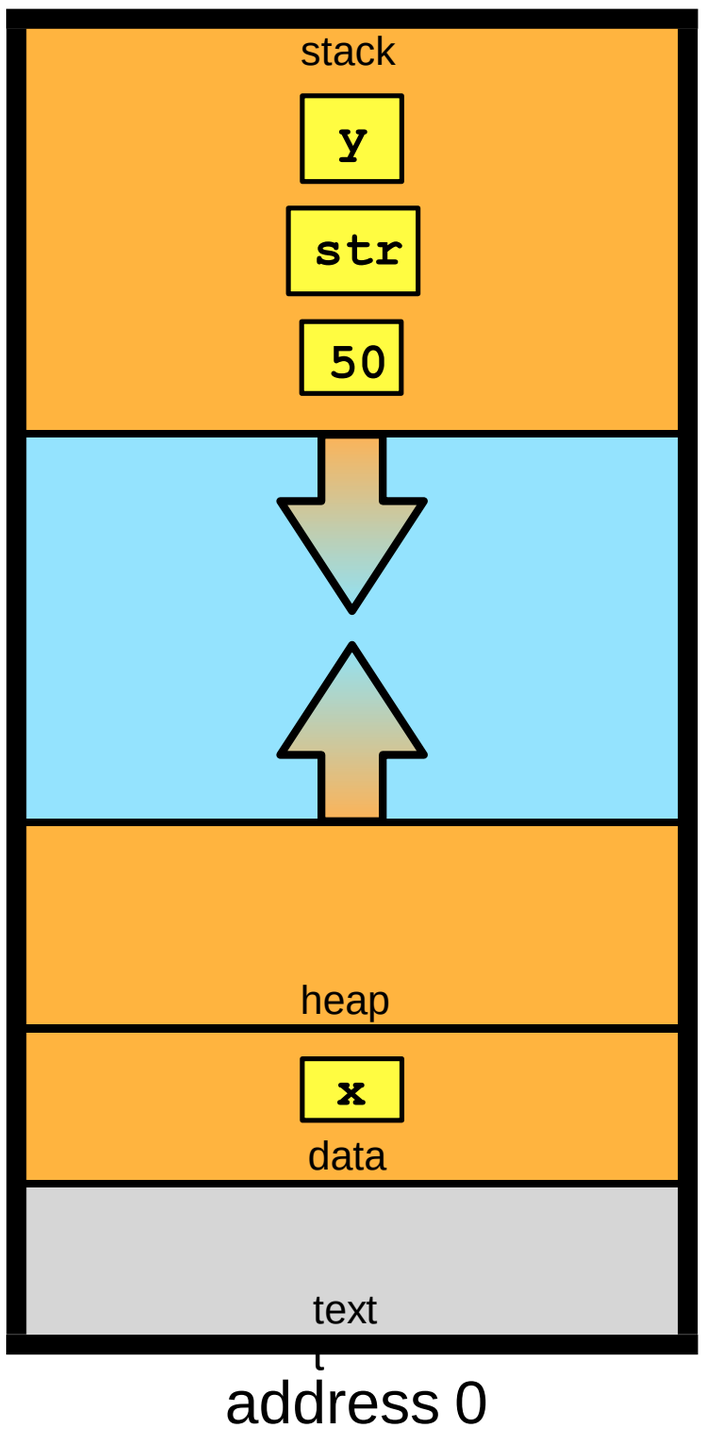
Running

When the process is selected to execute it changes state from ready to running.



Dynamic memory allocation

```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```



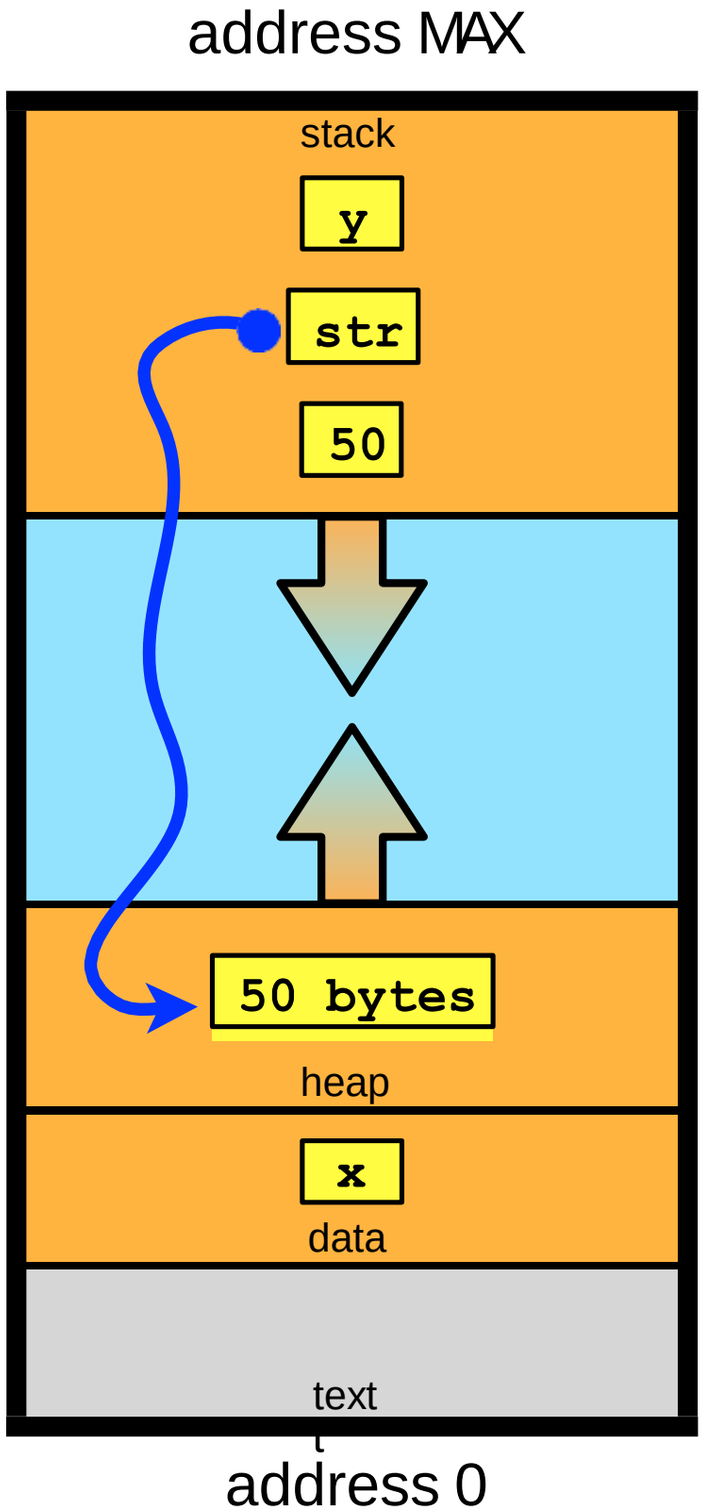
The size of the needed storage for the **local variables** `y` and `str` are also known at compile time but these are only needed within main and storage is allocated on the **stack**.

In general, the value of the **parameter** to the malloc might not be know at compile time. In this example the argument to malloc, the number 50 is pushed onto the **stack**.

Note: A compiler may also decide to put arguments in certain register.

Dynamic memory allocation

```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```

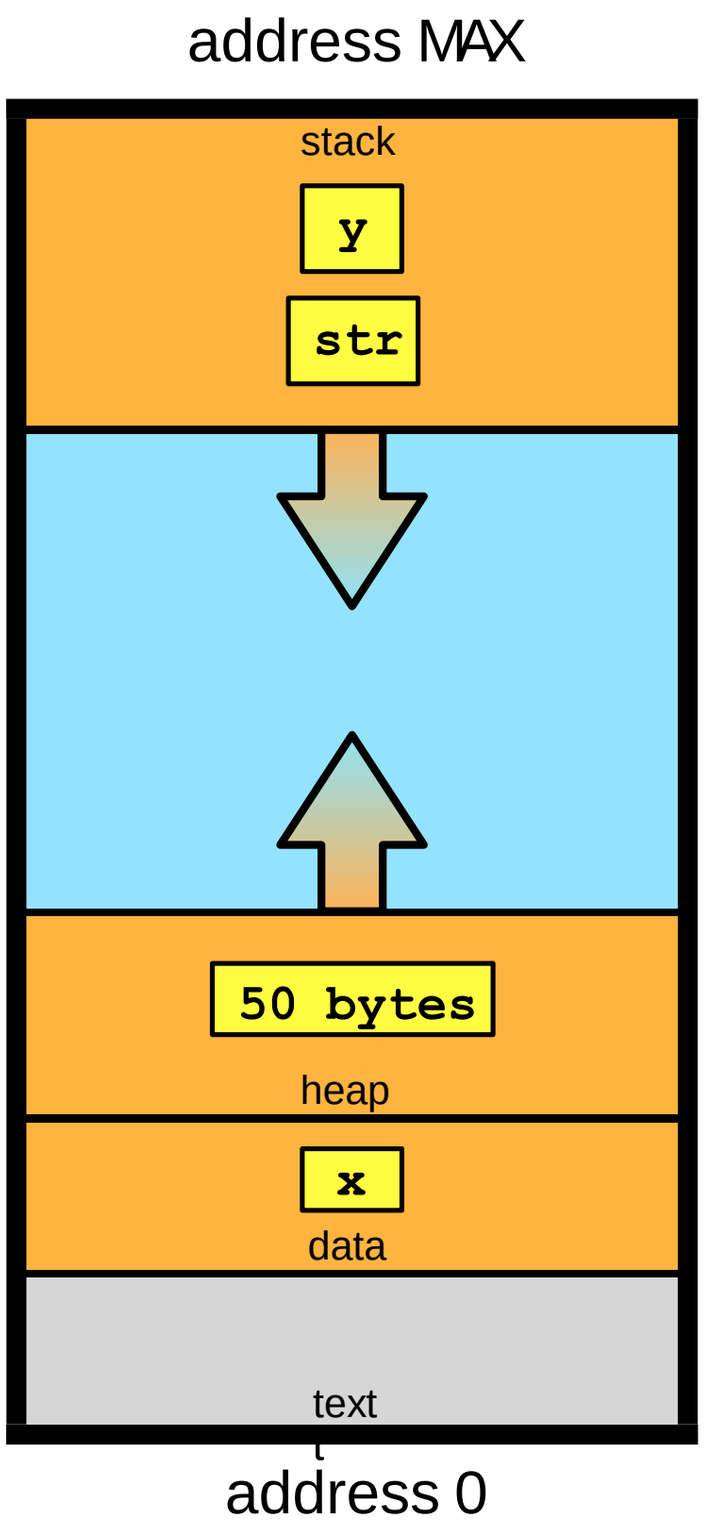


Now **`malloc`** has **dynamically** allocated memory on the heap.

The variable **`str`** stores the address to the first of the 50 bytes allocated on the heap, i.e., **`str`** is a **pointer**.

Dynamic memory deallocation

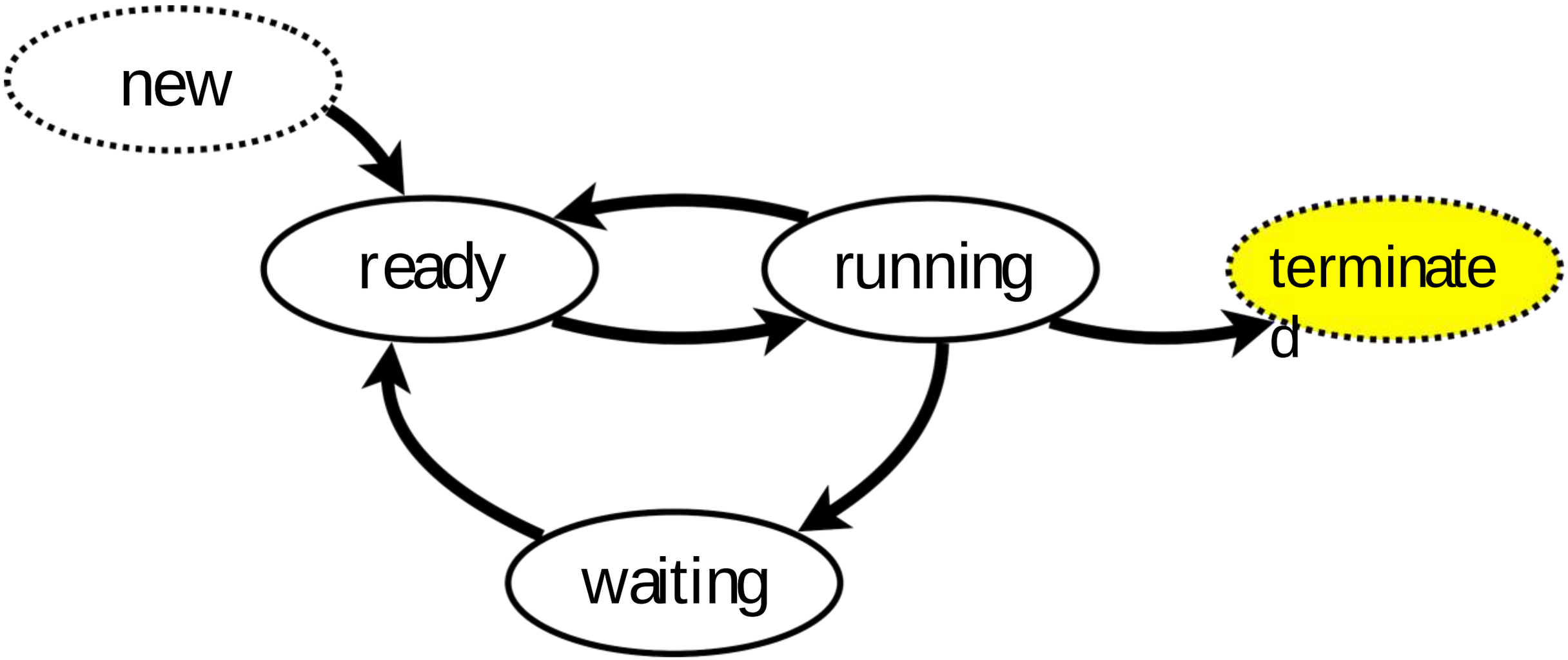
```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```



When malloc returns, the value 50, the argument used when calling malloc is no longer needed and is popped from the stack.

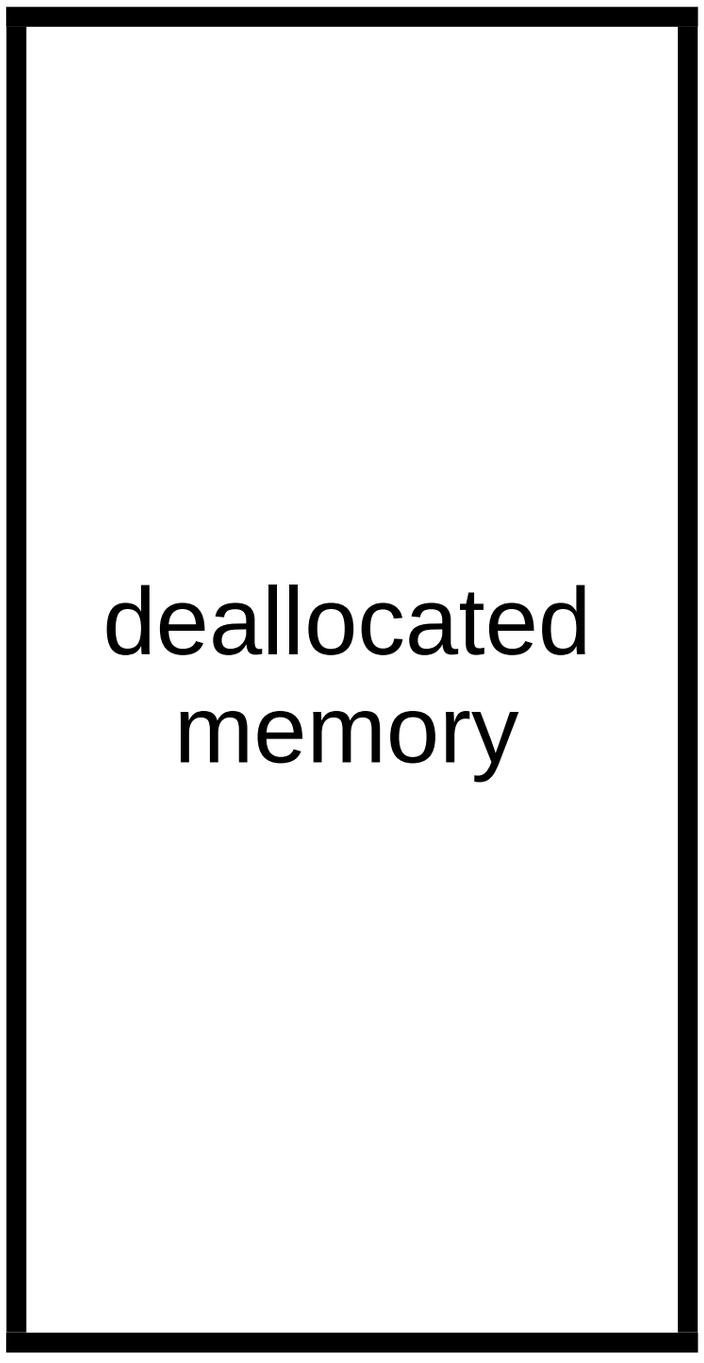
Termination

When the process terminates the operating system can deallocate the memory used by the process.



Process termination

```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```



The program terminates with a call to **`exit()`** and the memory allocated to the process can be deallocated.

Process Control Block (PCB)

Process Control Block (PCB)

The process control block (PCB) is a data structure in the operating system kernel containing the information needed to manage a particular process.

In brief, the PCB serves as the repository for any information that may vary from process to process.

Example of information stored in the PCB

Process Control Block (PCB)
Process id (PID)
Process state (new, ready, running, waiting or terminated)
CPU Context
I/O status information
Memory management information
CPU scheduling information

Context switch

Context switch

When CPU switches to another process, the system must save the state of the old process and restore the state for the new process via a **context switch**.

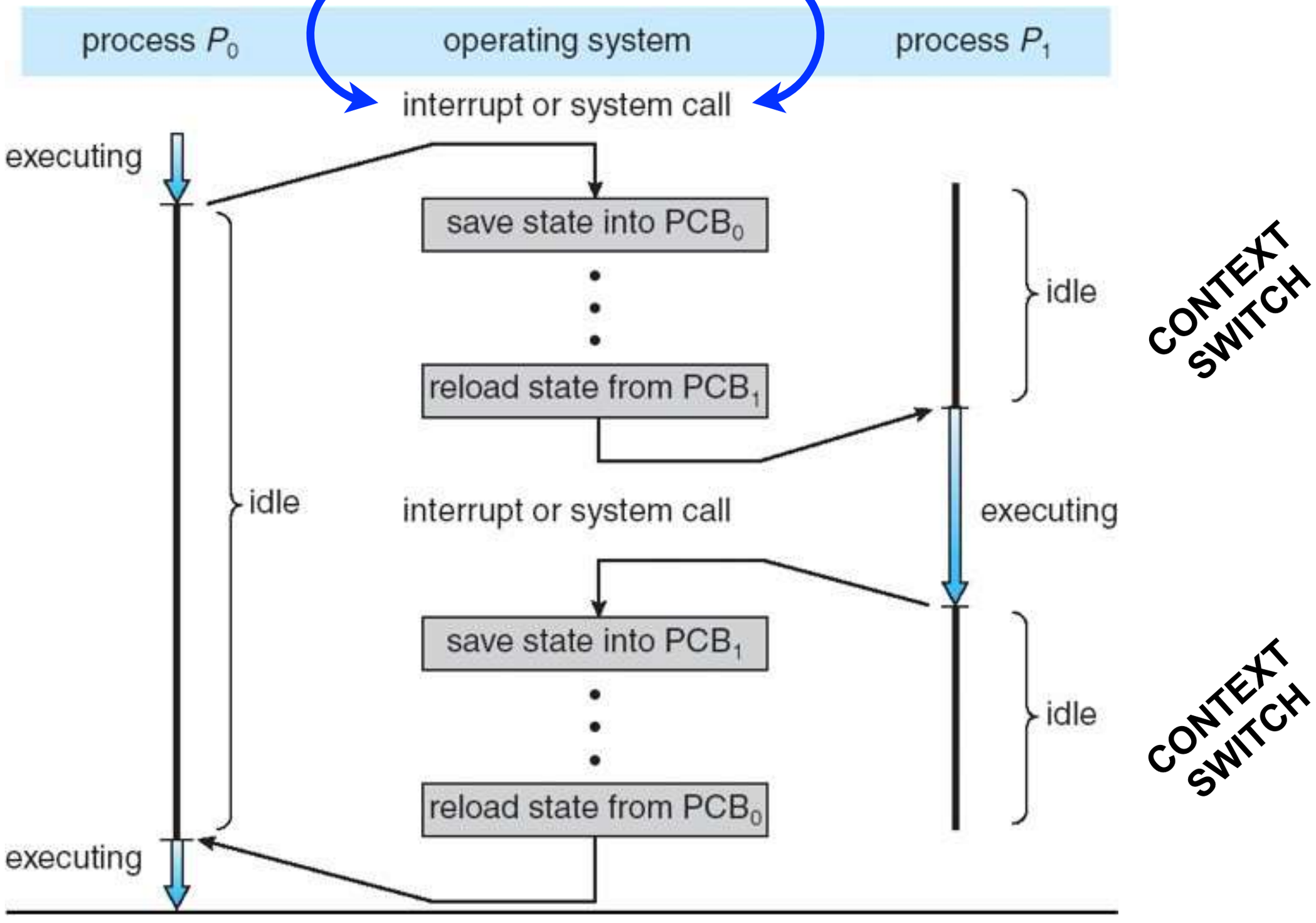
Context of a process is represented in the PCB.

Context-switch time is **overhead**; the system does no useful work while switching.

Time for context switch dependent on **hardware support**.

Exception or interrupt.

A system call is initiated using a system call exception.



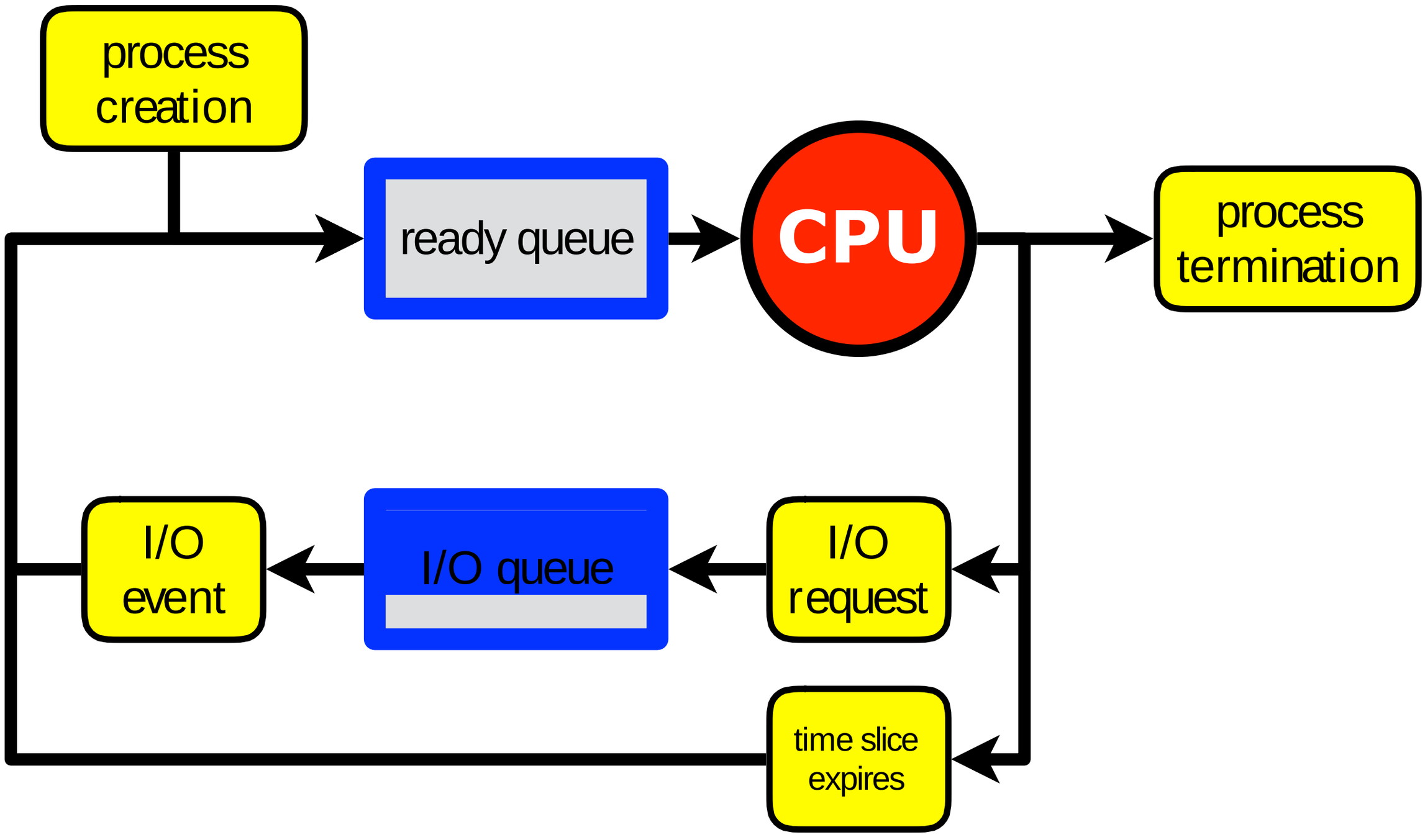
Process queues

Process queues

In general we may use one queue for each I/O device.

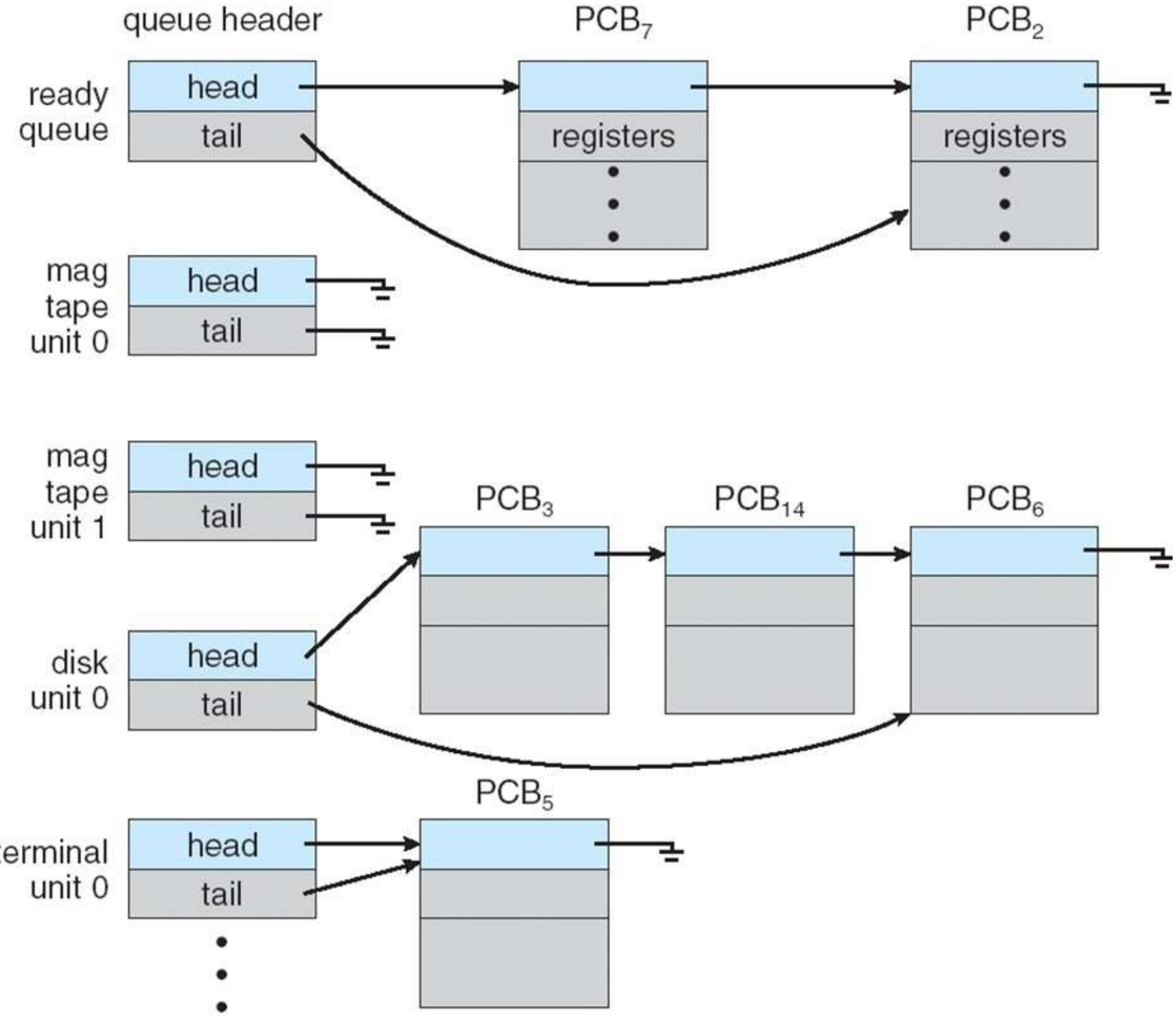
Ready queue	Set of all processes residing in main memory, ready and waiting to execute
Device queue	Set of processes waiting for an I/O device

Process queues



Process queues can be formed by linking PCBs together

Ready Queue

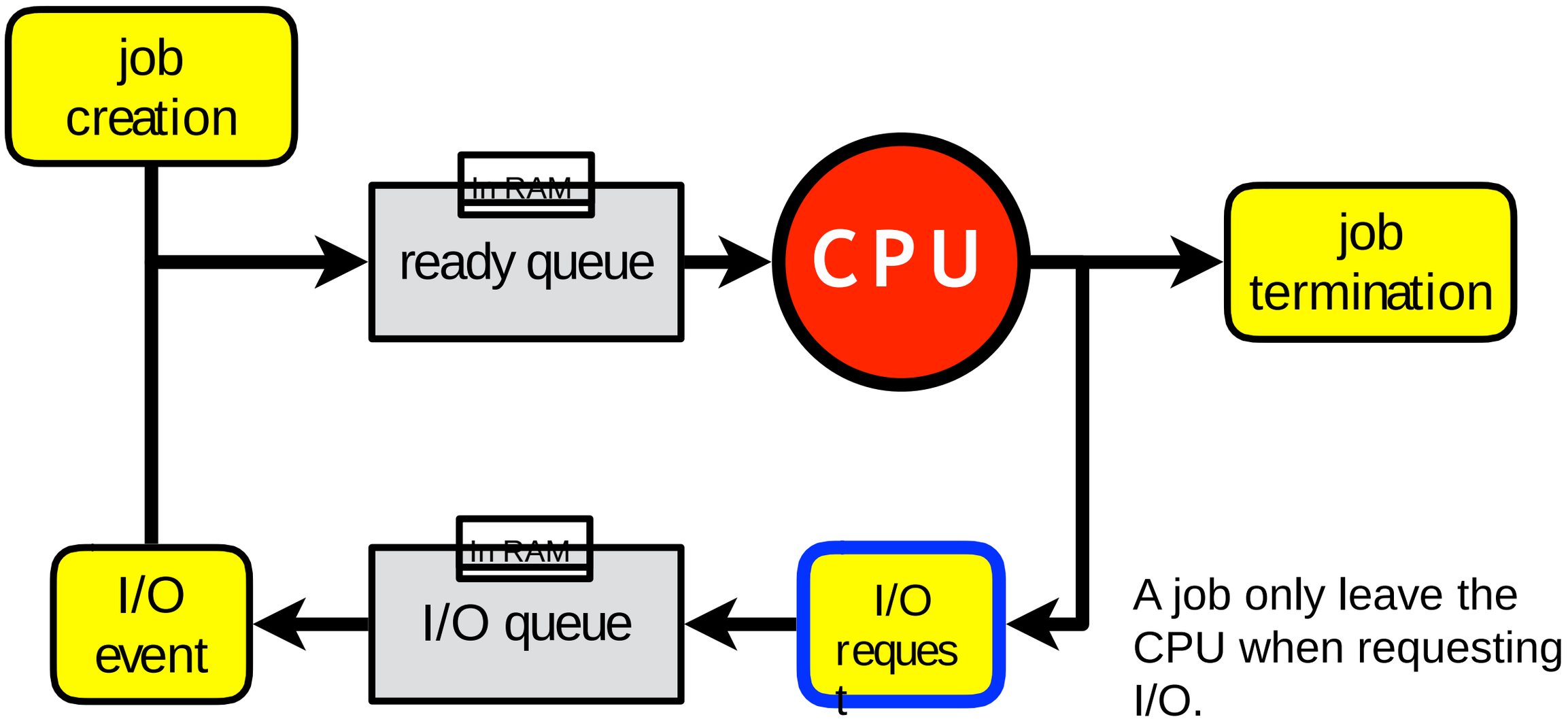


Device queues

CPU scheduling

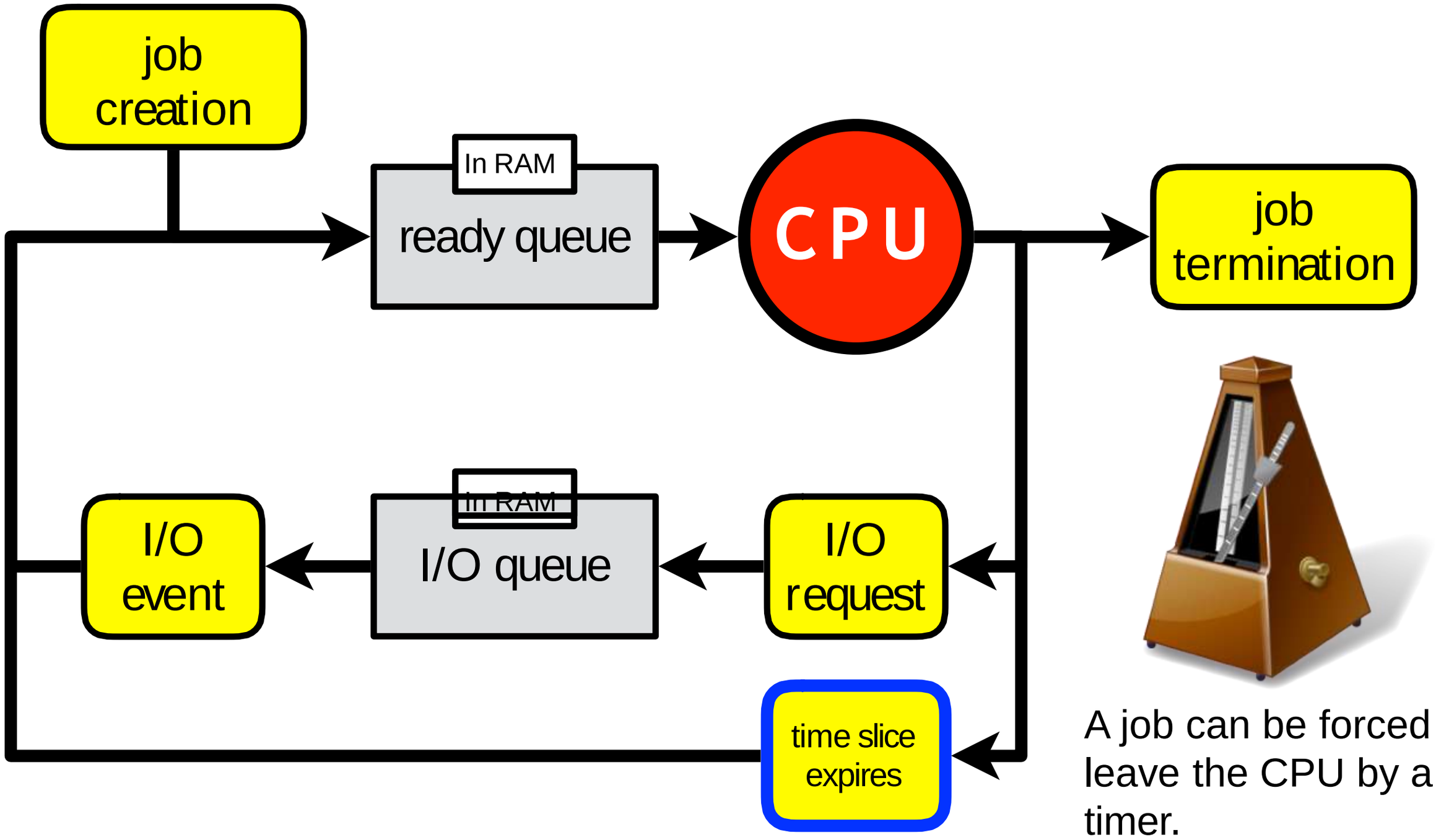
Multiprogramming

A schematic view of multiprogramming



Multitasking

A schematic view of multitasking

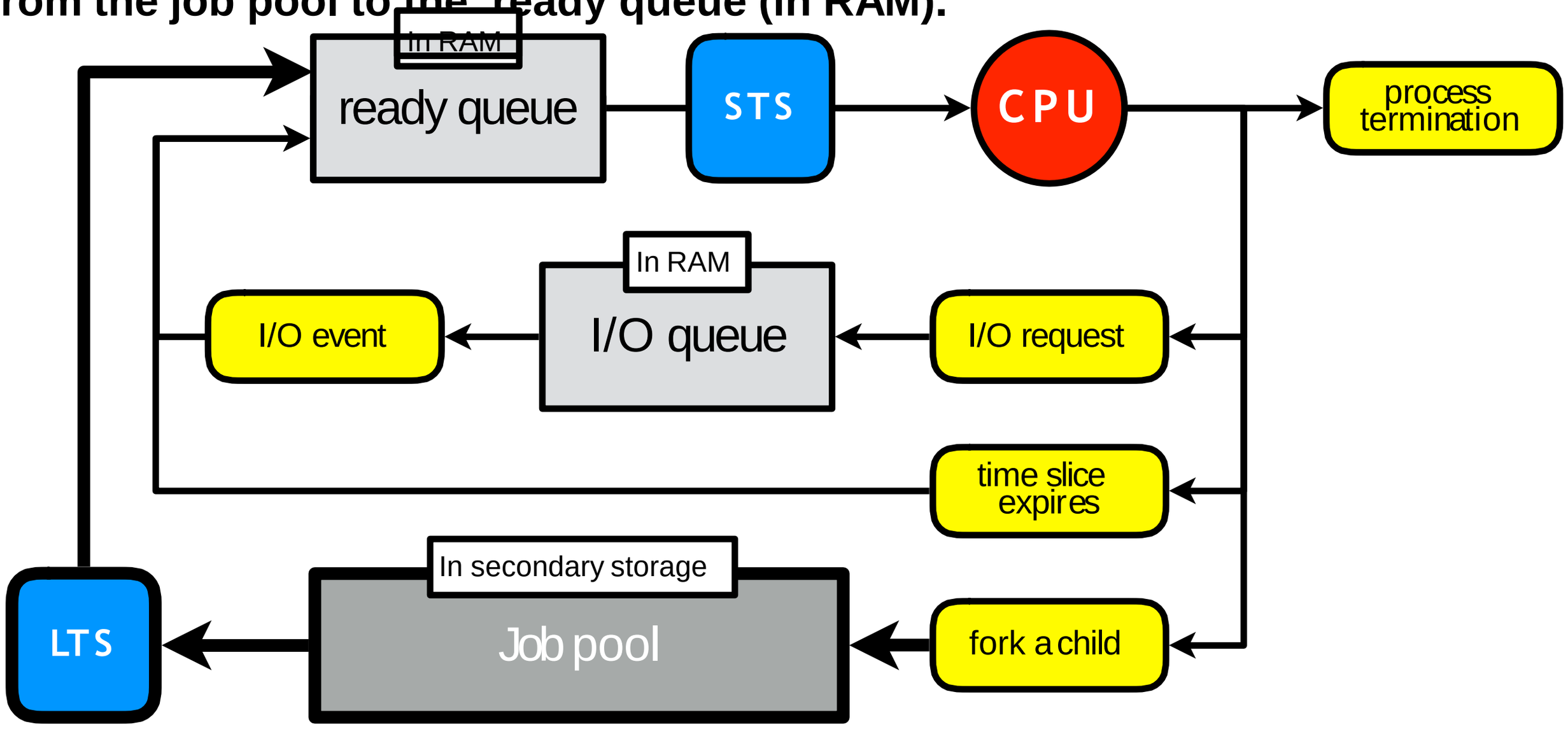


A job can be forced to leave the CPU by a timer.

Types of Schedulers

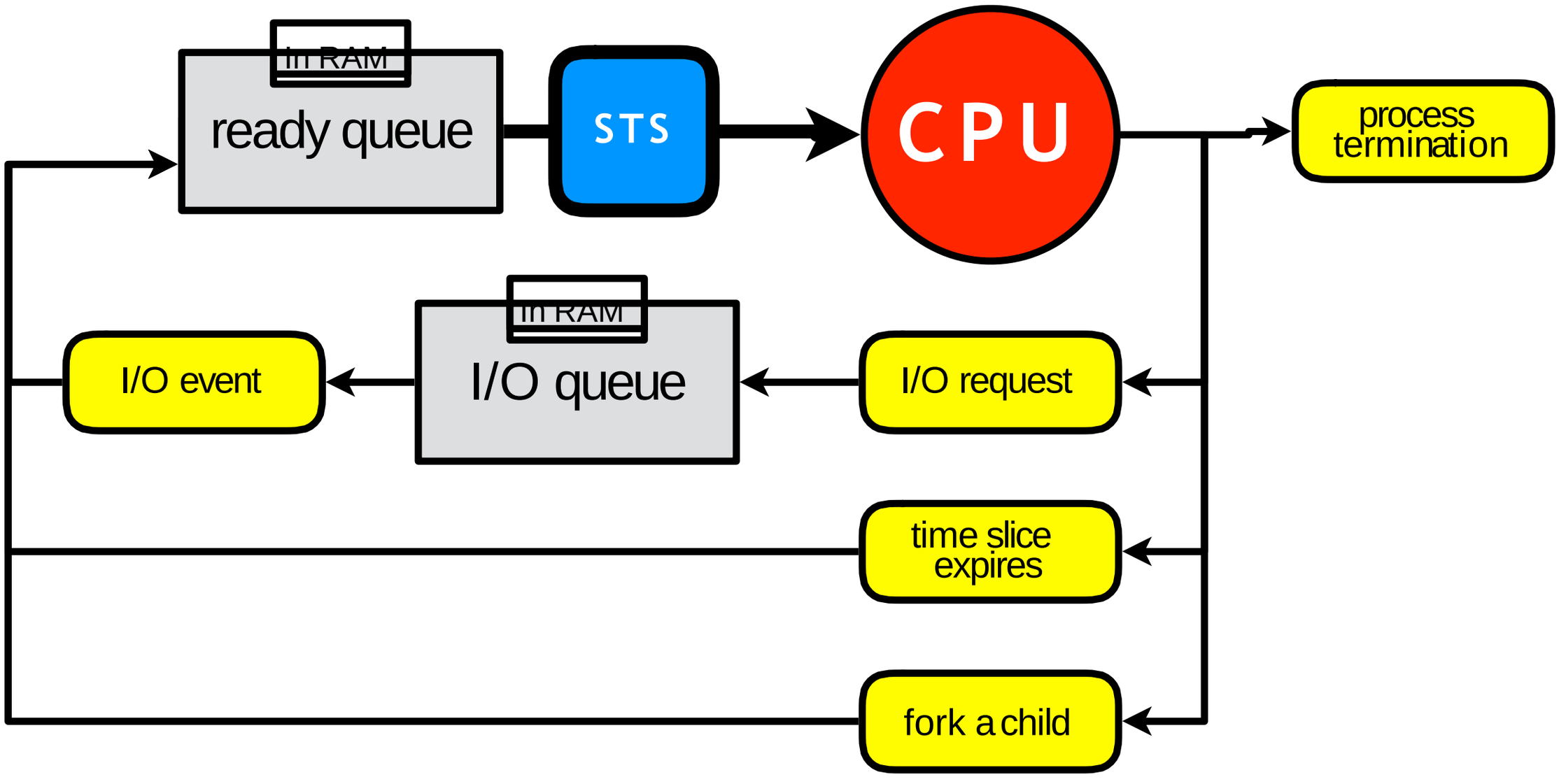
Long-term scheduler

The Long-term scheduler (LTS) (aka job scheduler) decides whether a new process should be brought into the ready queue in main memory or delayed. When a process is ready to execute, it is added to the job pool (on disk). When RAM is sufficiently free, some processes are brought from the job pool to the ready queue (in RAM).



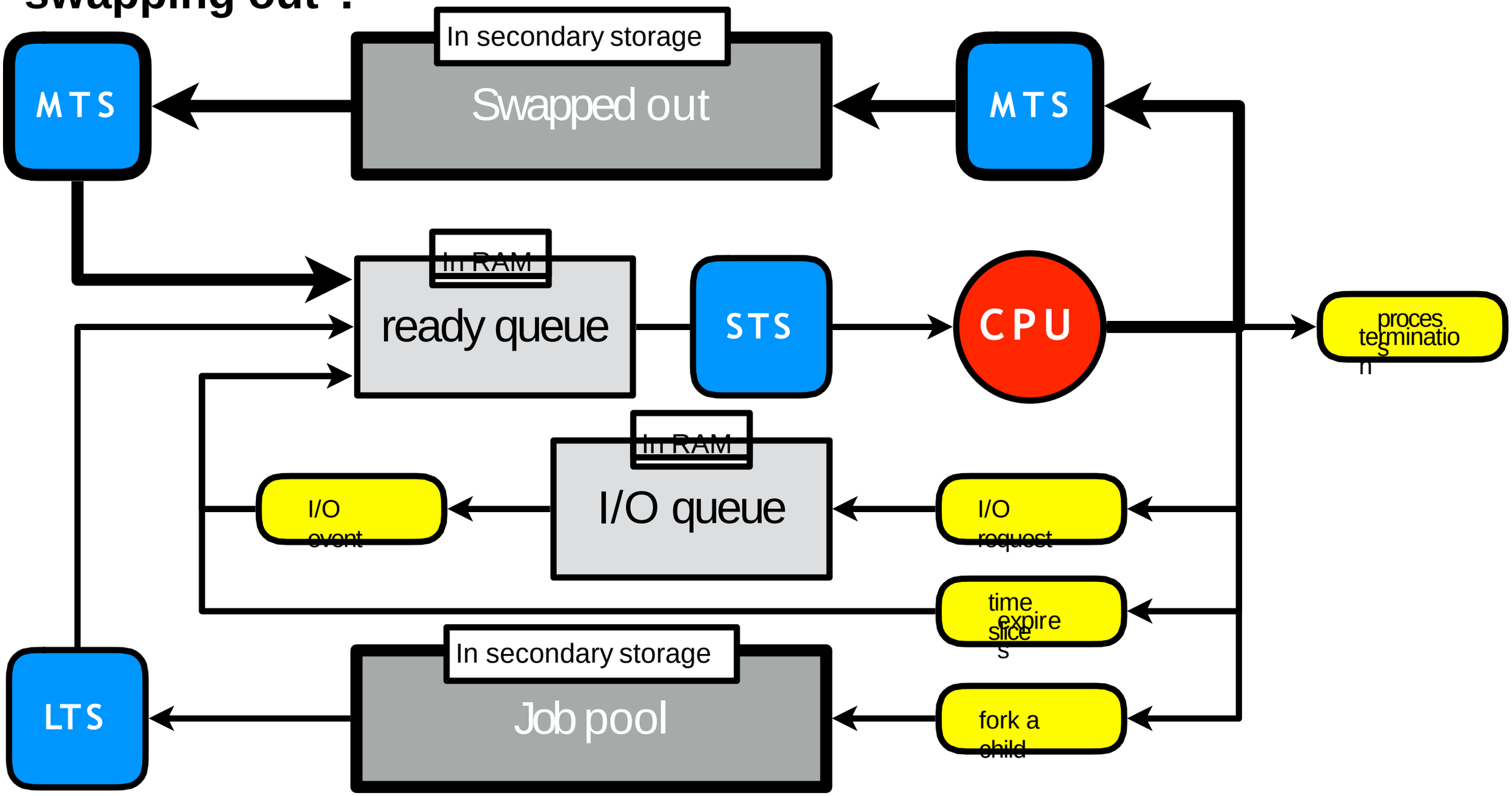
Short-term scheduler

The Short-term scheduler (STS), aka CPU scheduler, selects which process in the in memory ready queue that should be executed next and allocates CPU.



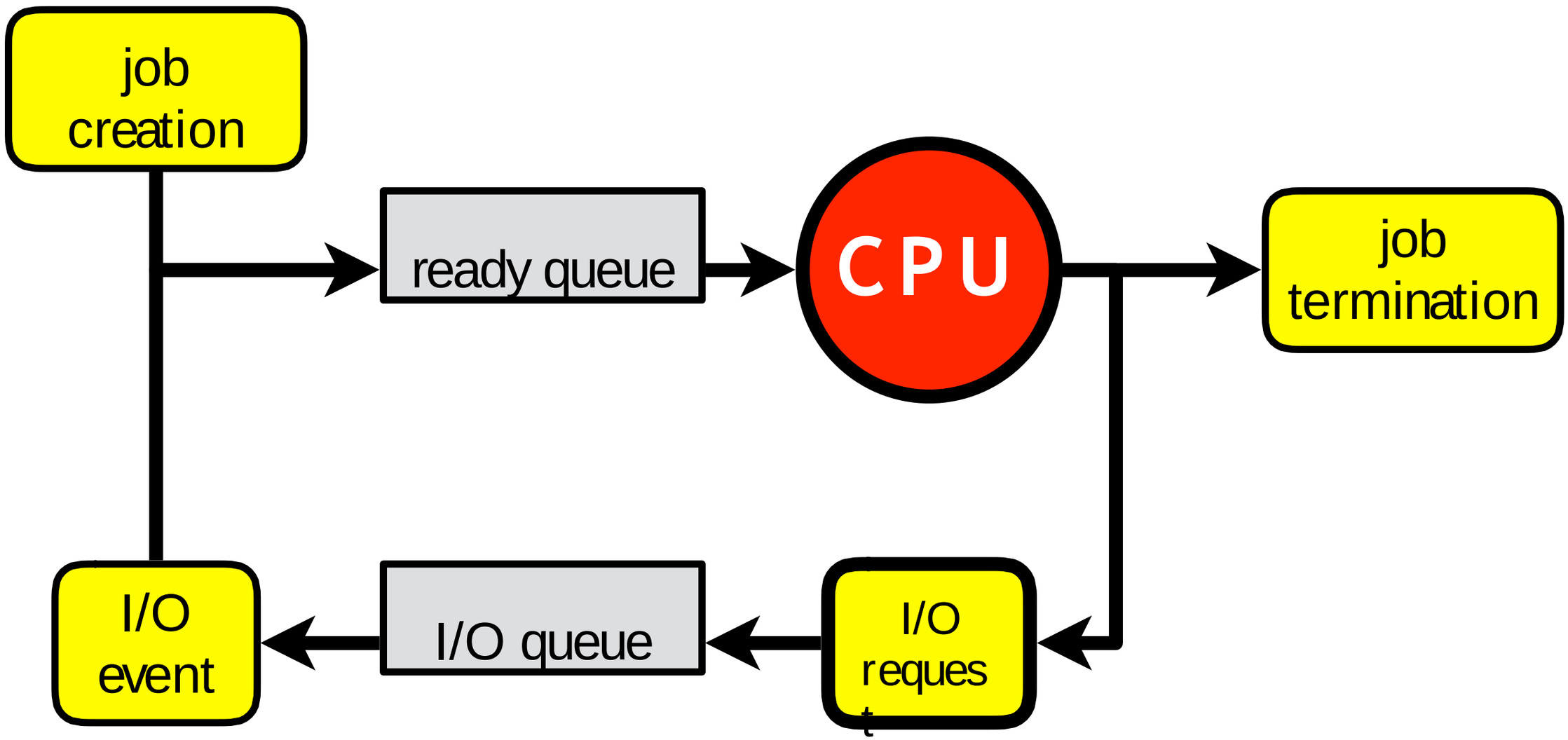
Medium-term scheduler

The medium-term scheduler (MTS) temporarily removes processes from main memory and places them in secondary storage and vice versa, which is commonly referred to as "swapping in" and "swapping out".



Multiprogramming

Multiprogramming **maximises CPU utilisation**.

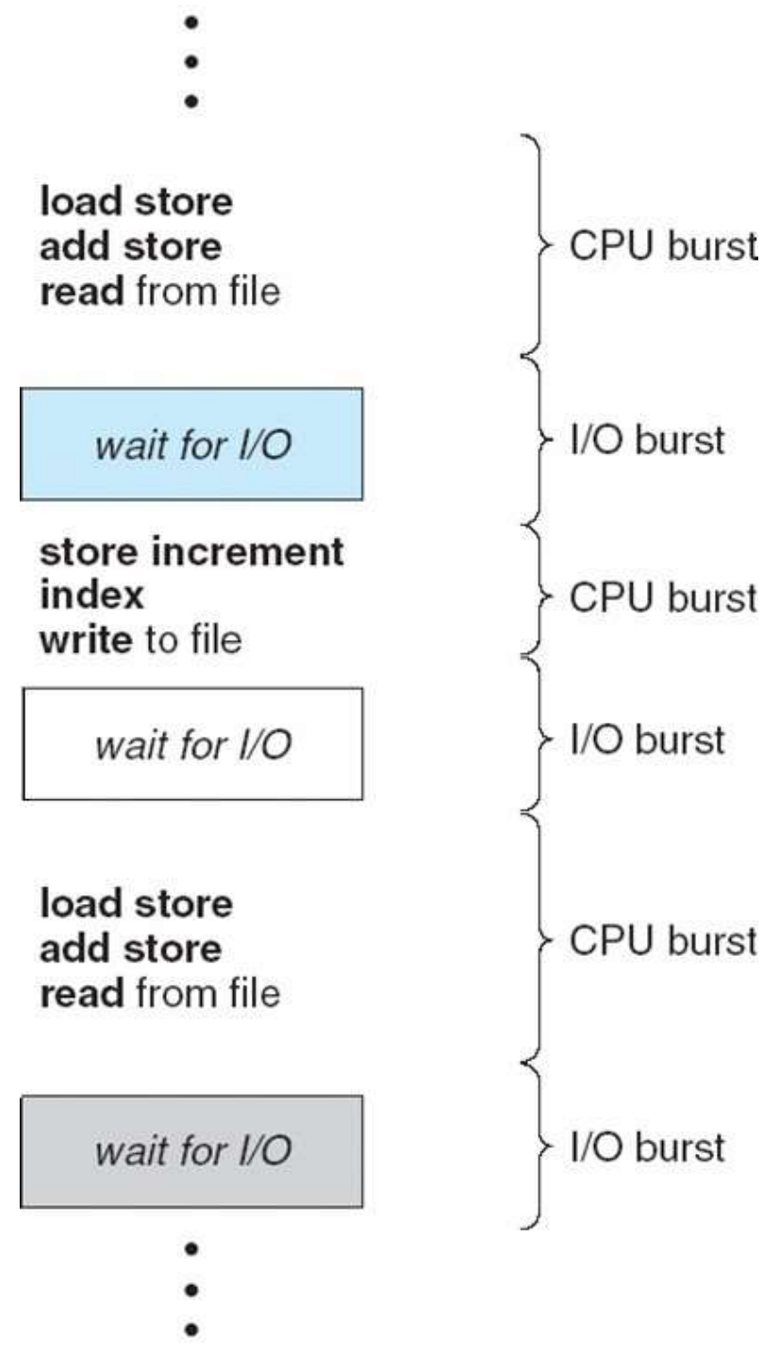


Scheduling criteria

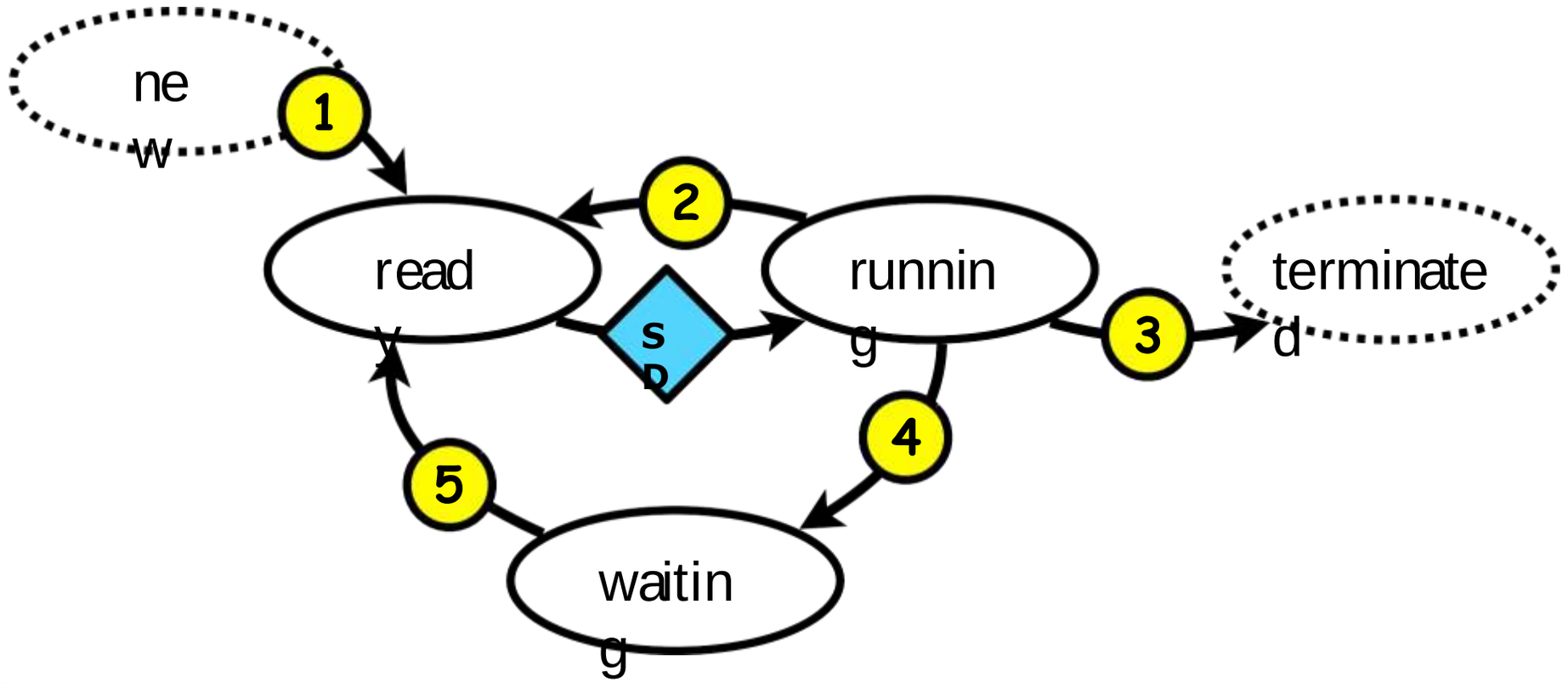
Criteria	Definition	Goal
CPU utilization	The % of time the CPU is executing user level process code.	Maximize
Throughput	Number of processes that complete their execution per time unit.	Maximize
Turnaround time	Amount of time to execute a particular process.	Minimize
Waiting time	Amount of time a process has been waiting in the ready queue.	Minimize
Response time	Amount of time it takes from when a request was submitted until the first	Minimize

CPU bursts and I/O bursts

When a program executes it alternates between CPU bursts and I/O bursts.

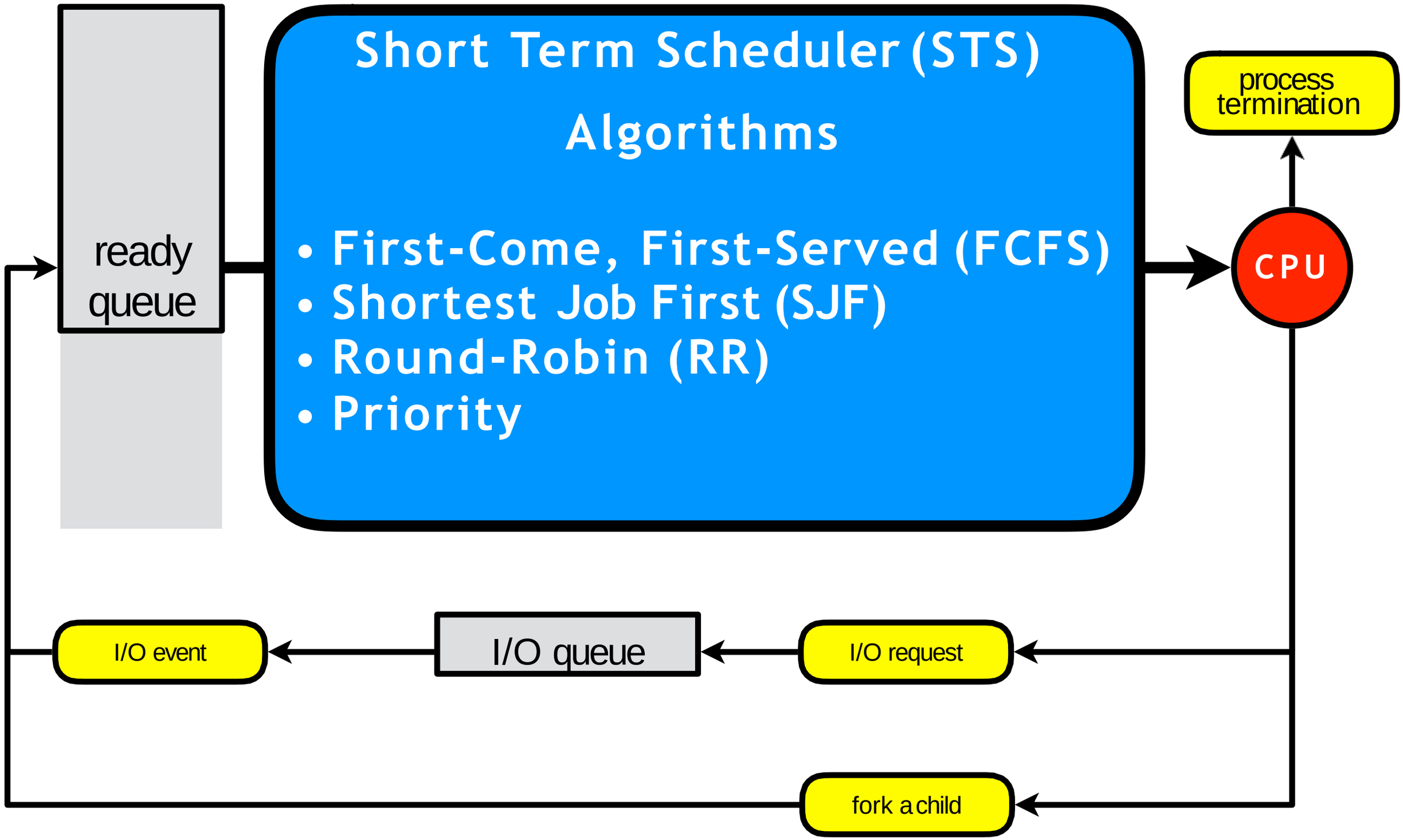


Scheduling algorithms



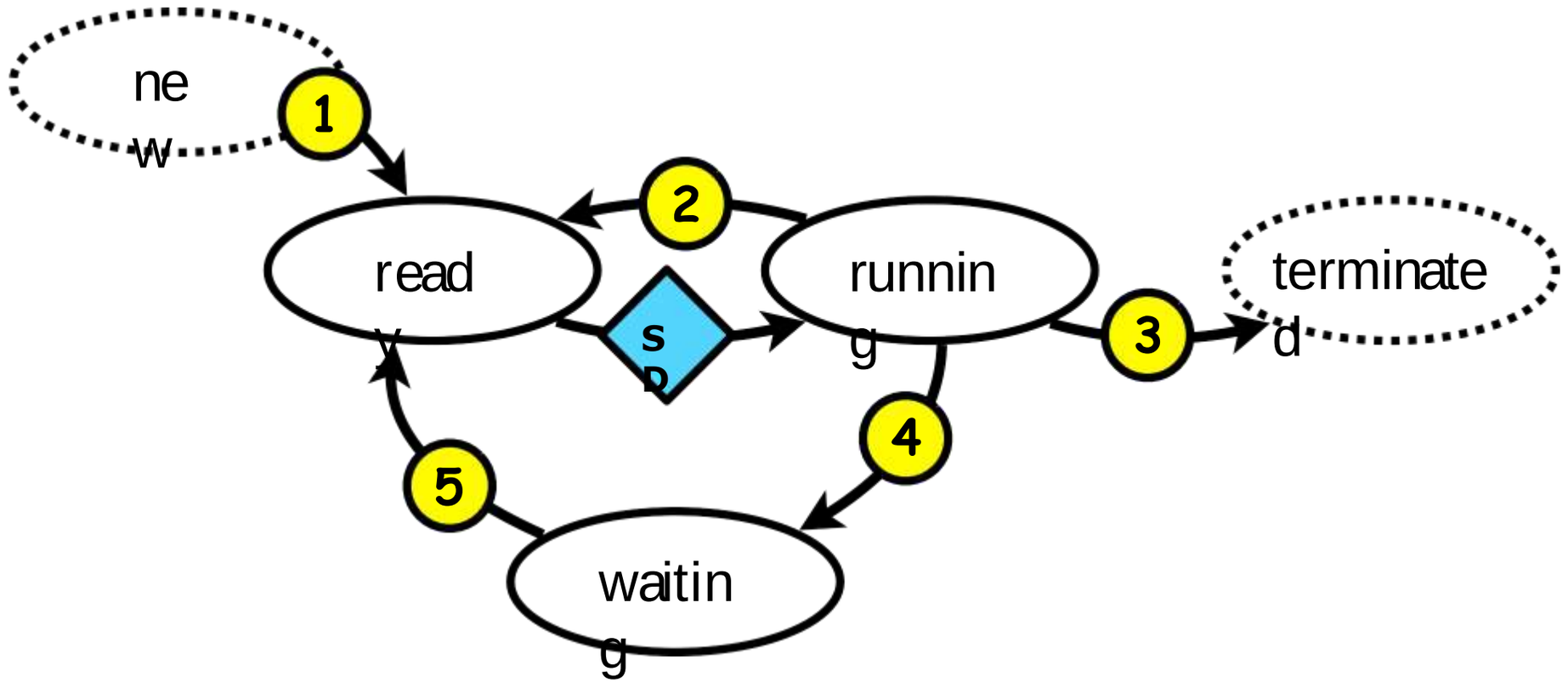
1. A new process arrives at the ready queue.
2. Time slice expires or other forms of **preemption**.
3. The running process terminates.
4. The running process requests I/O.
5. An I/O request completes.

Scheduling algorithms



Preemptive and nonpreemptive

Scheduler dispatch can be **preemptive** or **nonpreemptive**.

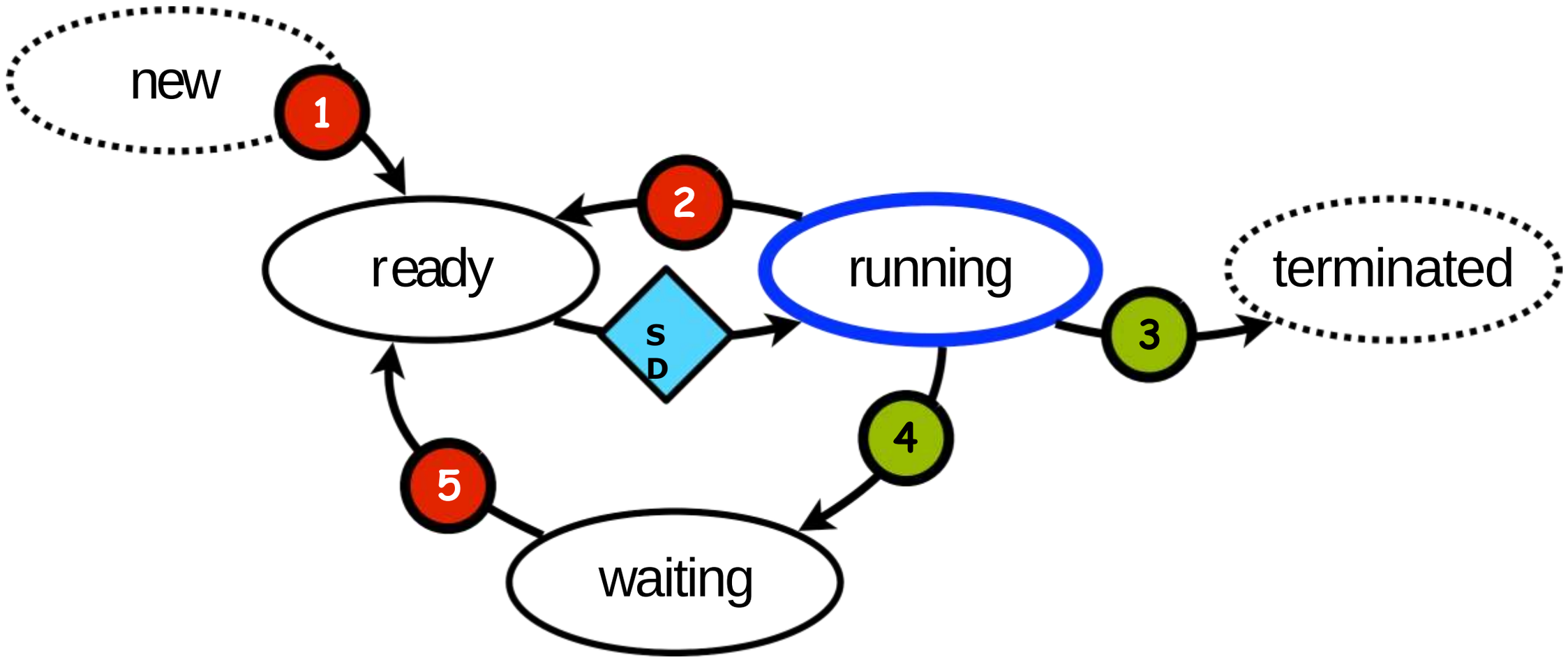


A **preemptive** dispatch is caused by an event external to the running running process.

A **nonpreemptive** dispatch is caused by the running process itself.

Preemptive and nonpreemptive

Scheduler dispatch can be **preemptive** or **nonpreemptive**.

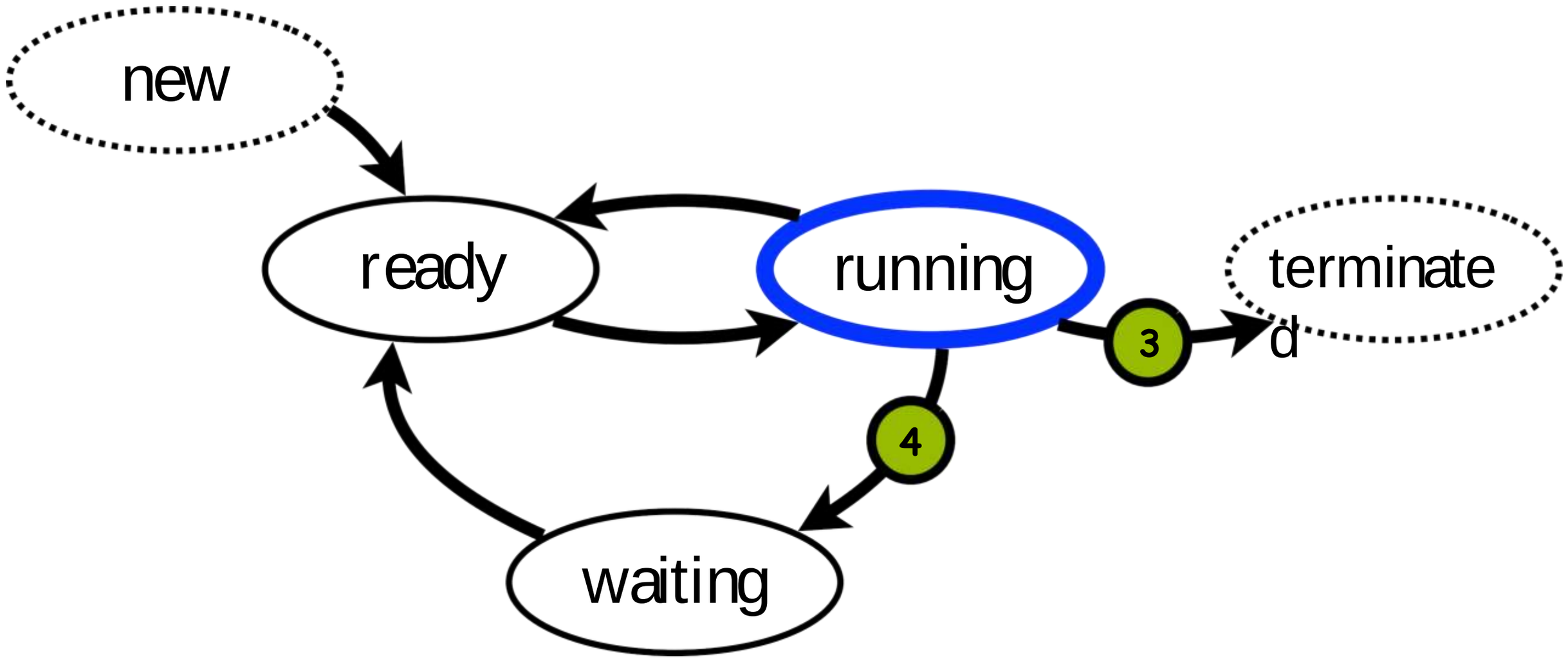


Events causing a **preemptive** scheduler dispatch: 1 2 5

Events causing a **nonpreemptive** scheduler dispatch: 3 4

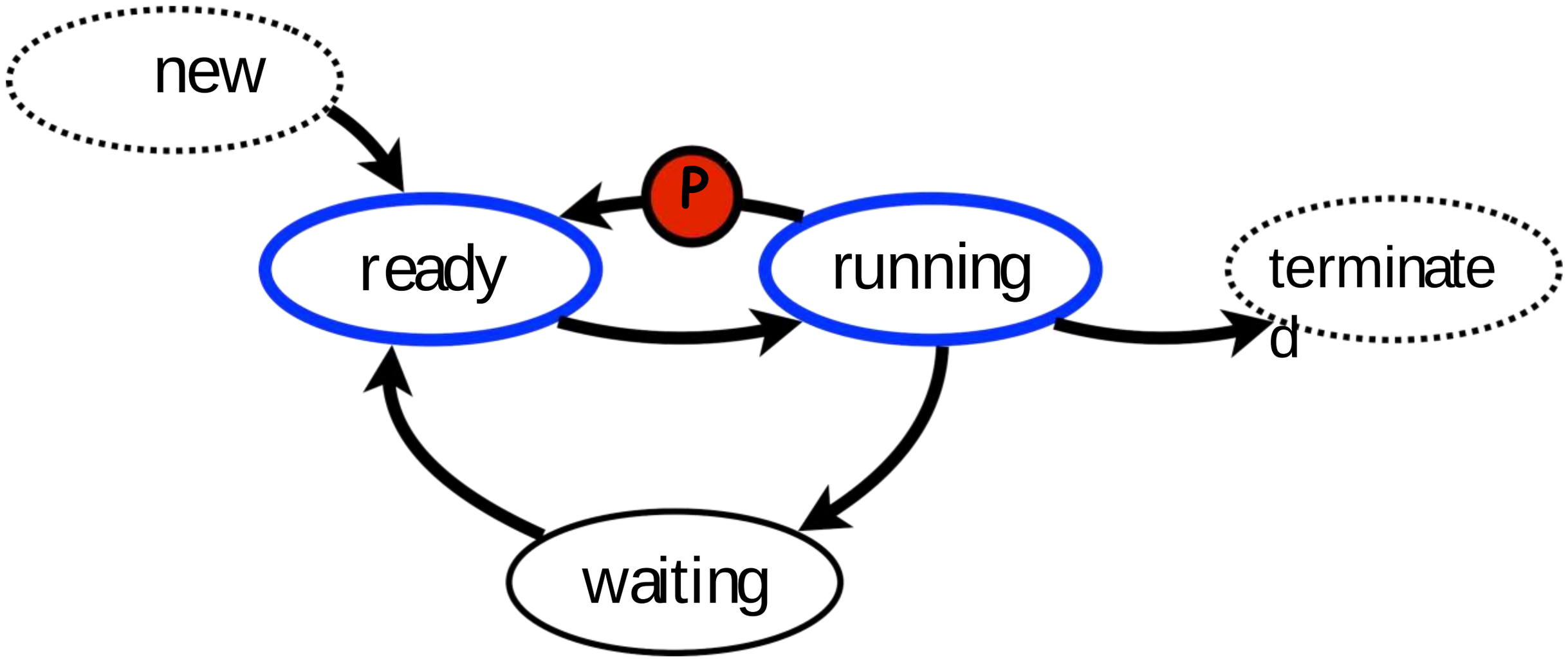
CPU burst

With CPU burst we mean the time spent by a running process using the CPU before (3) terminating or (4) performing a blocking system call.



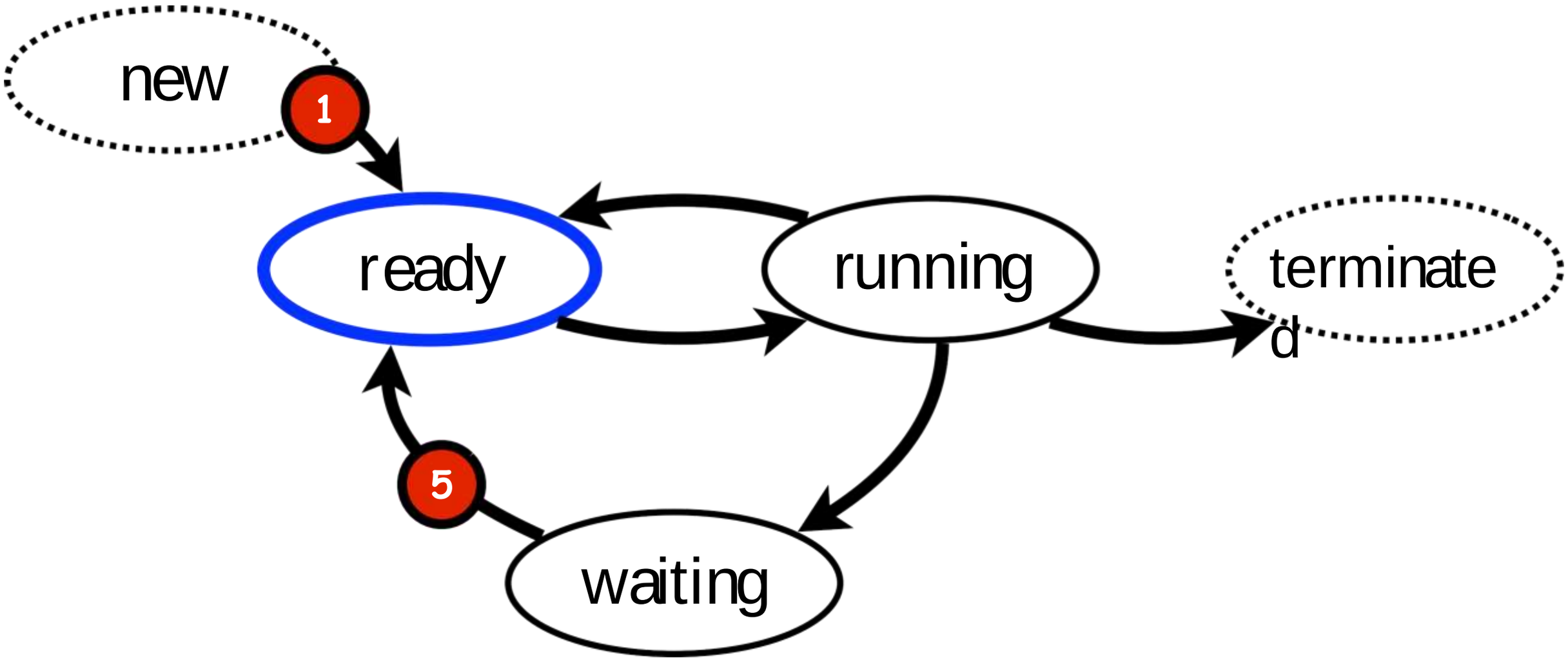
Preemption

A process may get preempted (P) i.e., forced off the CPU and put back in the ready queue before completing its CPU burst.



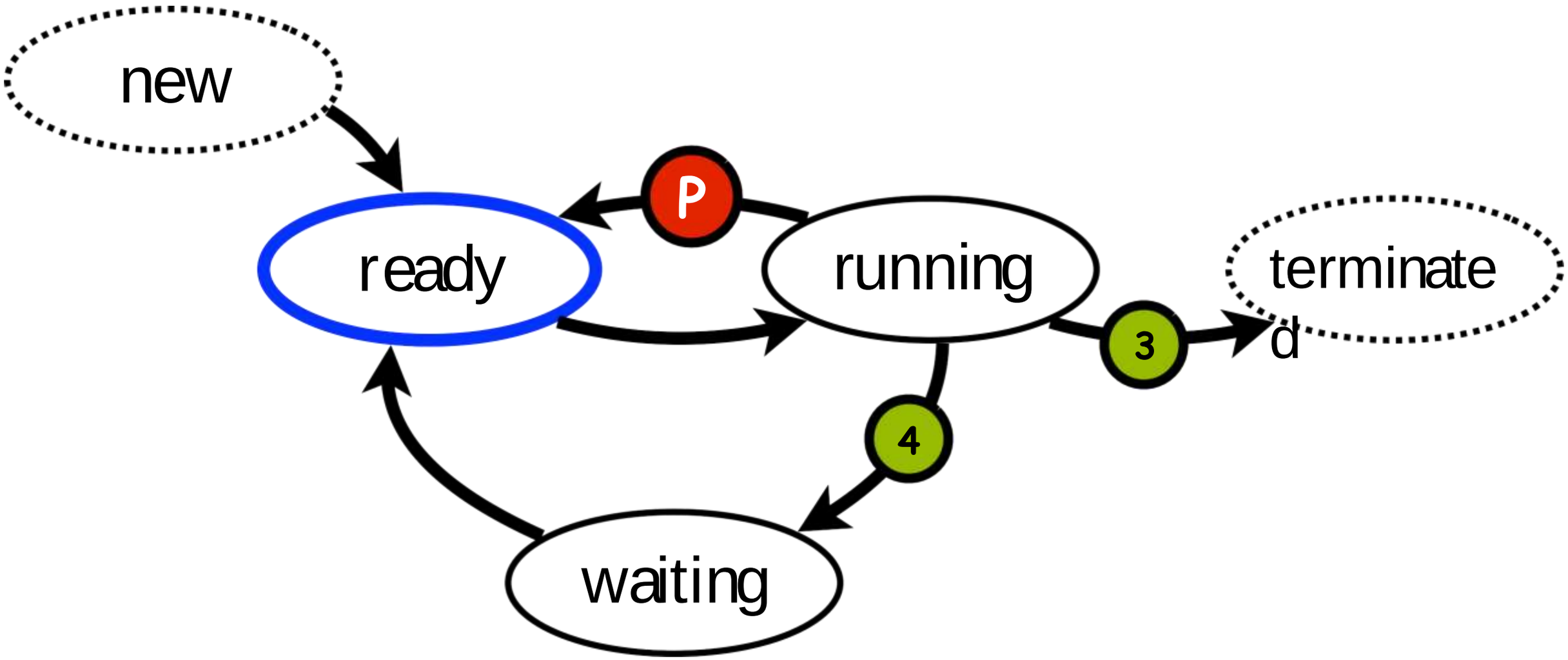
Process arrival

We make no difference between 1 a new process arriving to the ready queue and 5 a process coming back to the ready queue after completion of a blocking system call.



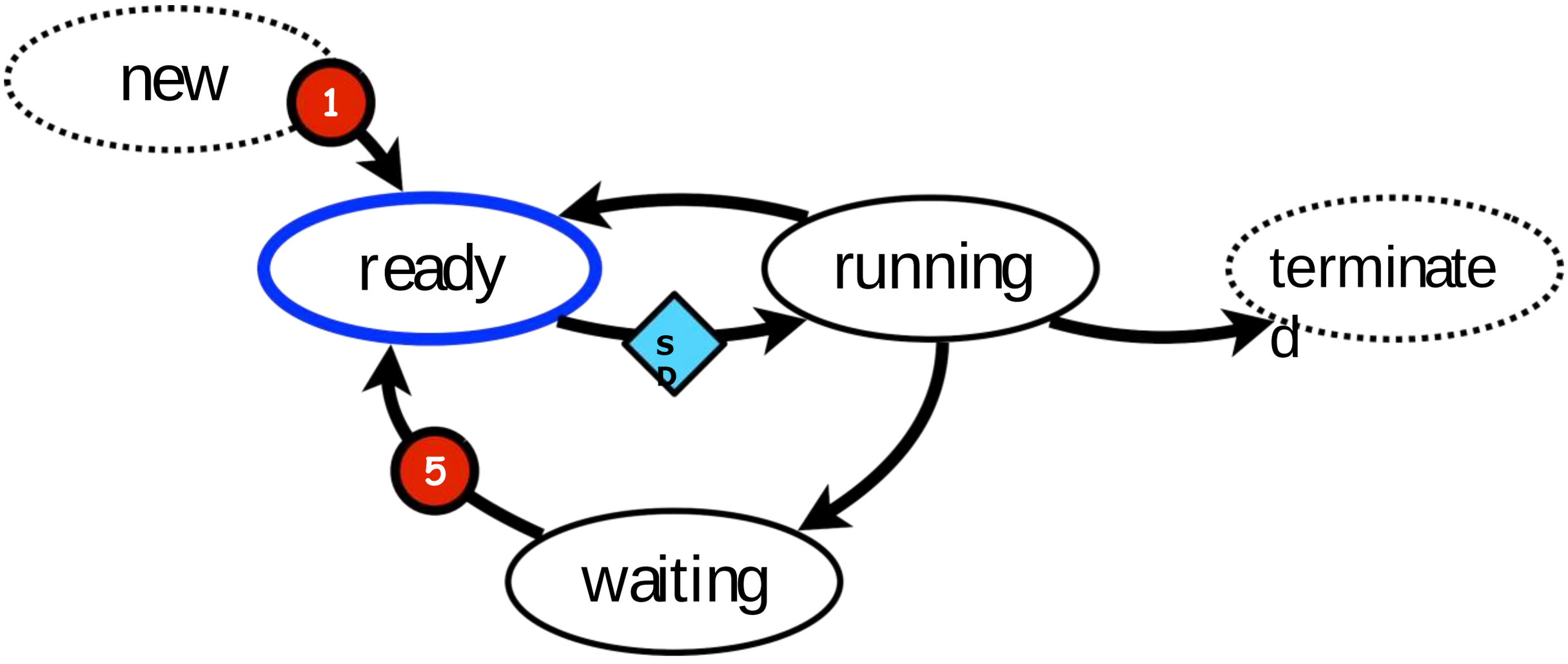
Waiting time

With waiting time we mean the total time a process has been waiting in the ready queue until 3 terminating or 4 performing a blocking system call. A process may get preempted P and forced off the CPU and put back in the ready queue before completing its CPU burst and have more waiting time added.



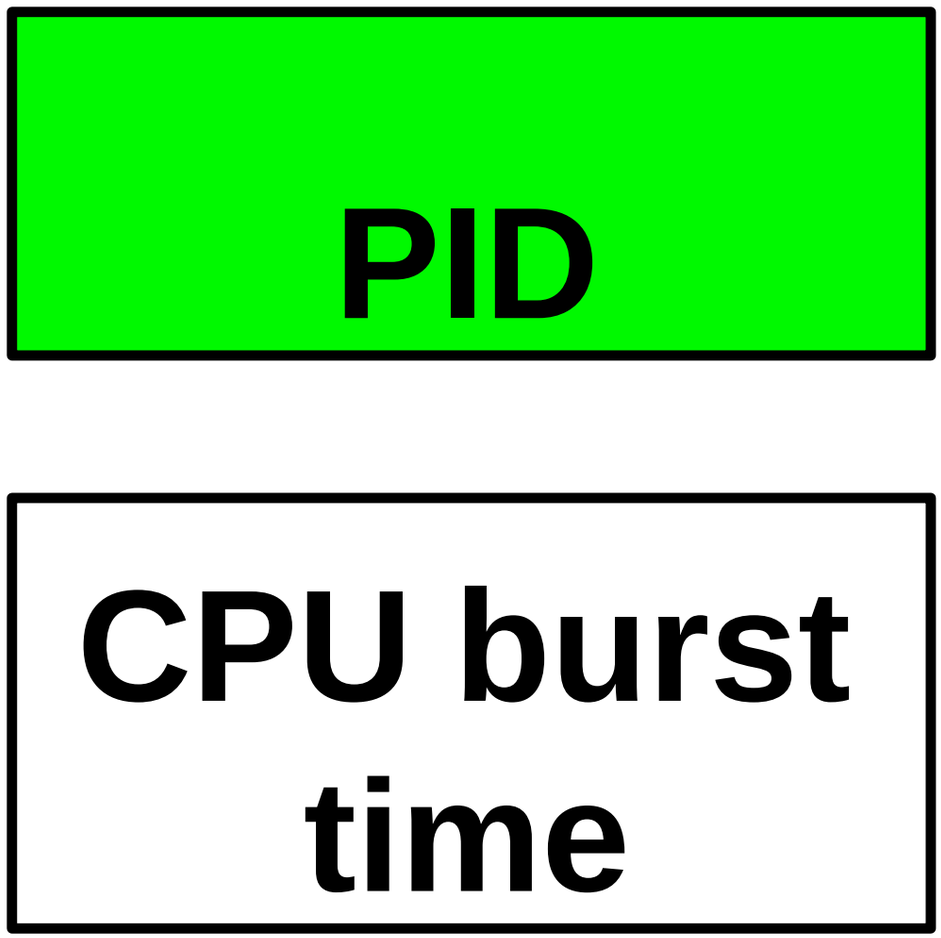
Response time

The model we use to study and compare different CPU scheduling algorithms is abstract and don't take into account what a response is and that it may take time to produce a response once a task gets to execute on the CPU. In this model response time is defined as the time from when a task enters the ready queue (1 or 5) to the time the task first gets to execute on the cpu SD .



Process representation

When studying CPU scheduling a process will be represented by process ID (PID) and the next CPU burst time.

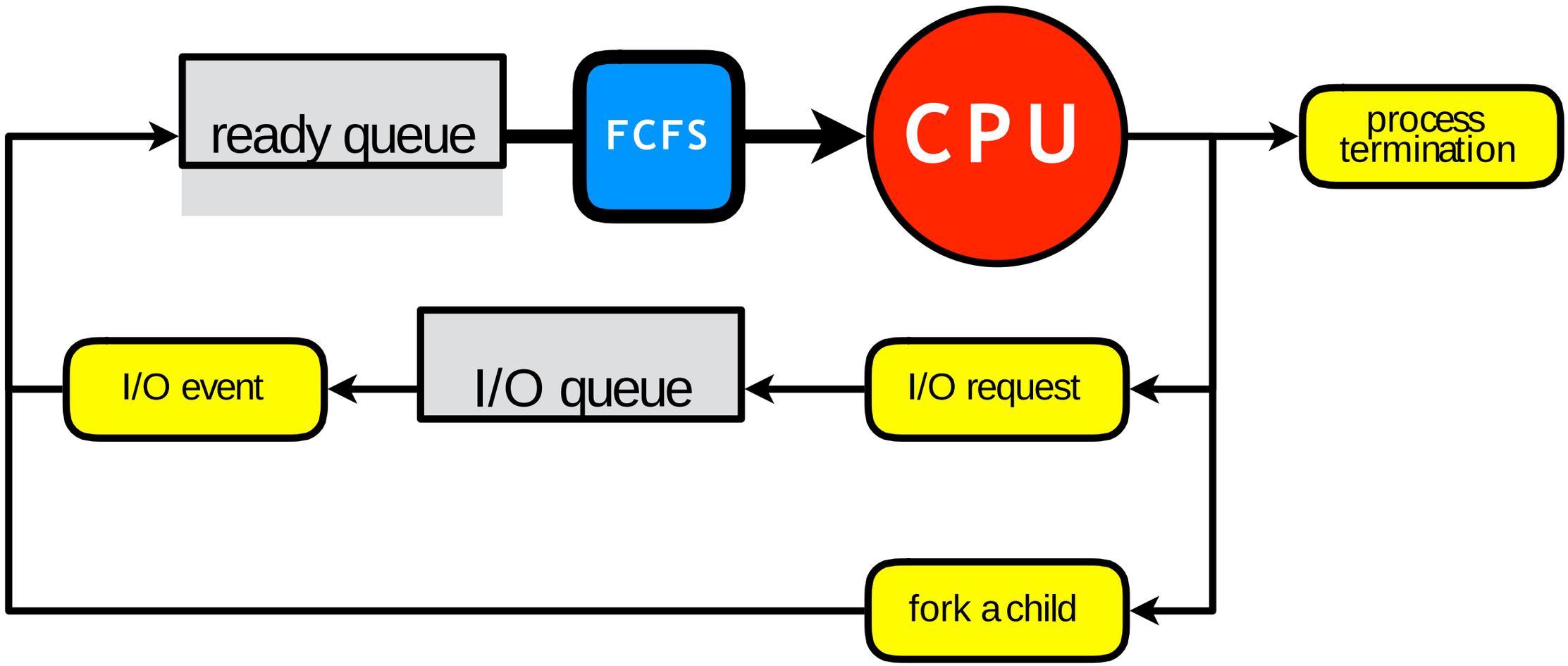


FCFS

First-Come, First-Served

First-Come, First-Served (FCFS)

The first come, first served (commonly called FIFO – first in, first out) process scheduling algorithm is the simplest process scheduling algorithm. Processes are executed on the CPU in the same order they arrive to the ready queue.



The convoy effect

When using FCFS scheduling, if I/O bound (short CPU burst) processes are scheduled after CPU bound (long CPU burst) processes, the average waiting time increases.

Short CPU burst

Short CPU burst

Long CPU burst

First-Come, First-Served (FCFS)

Advantages

- ★ Simple
- ★ Easy
- ★ First come, first served

Disadvantages

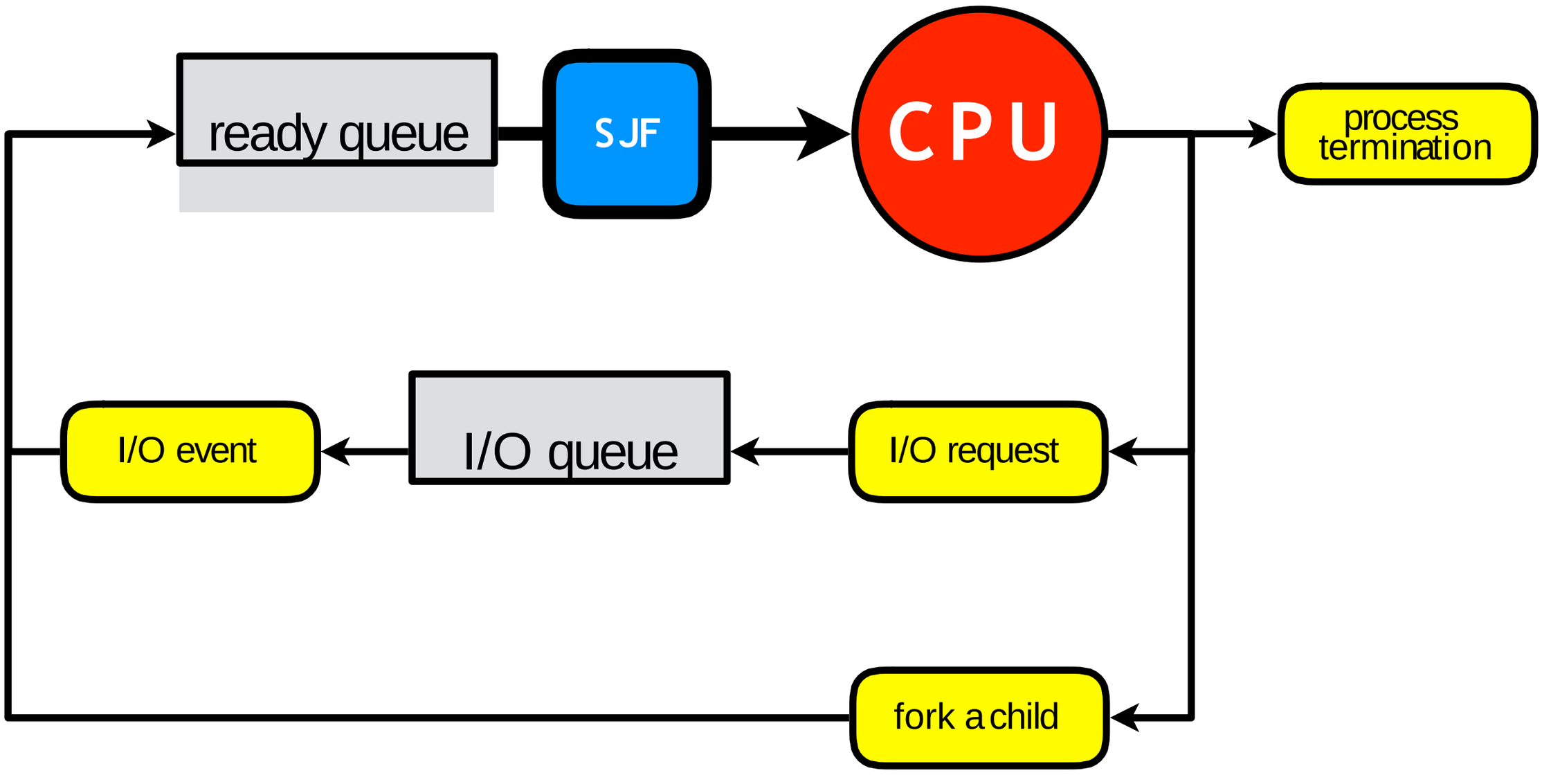
- ★ This scheduling method is **nonpreemptive**, that is, a process will execute its CPU burst until it finishes.
- ★ Because of this nonpreemptive scheduling, short processes which are at the back of the queue have to wait for the long process at the front to finish making the average waiting time increase - the **convoy effect**.

SJF

Shortest Job First

Shortest Job First (SJF)

Shortest Job First (SJF) scheduling assigns the process estimated to complete fastest, i.e, the process with shortest CPU burst, to the CPU as soon as CPU time is available.



Shortest Job First (SJF)

Shortest Job First (SJF) scheduling assigns the process estimated to complete fastest to the CPU as soon as CPU time is available.

- ★ Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest burst time.

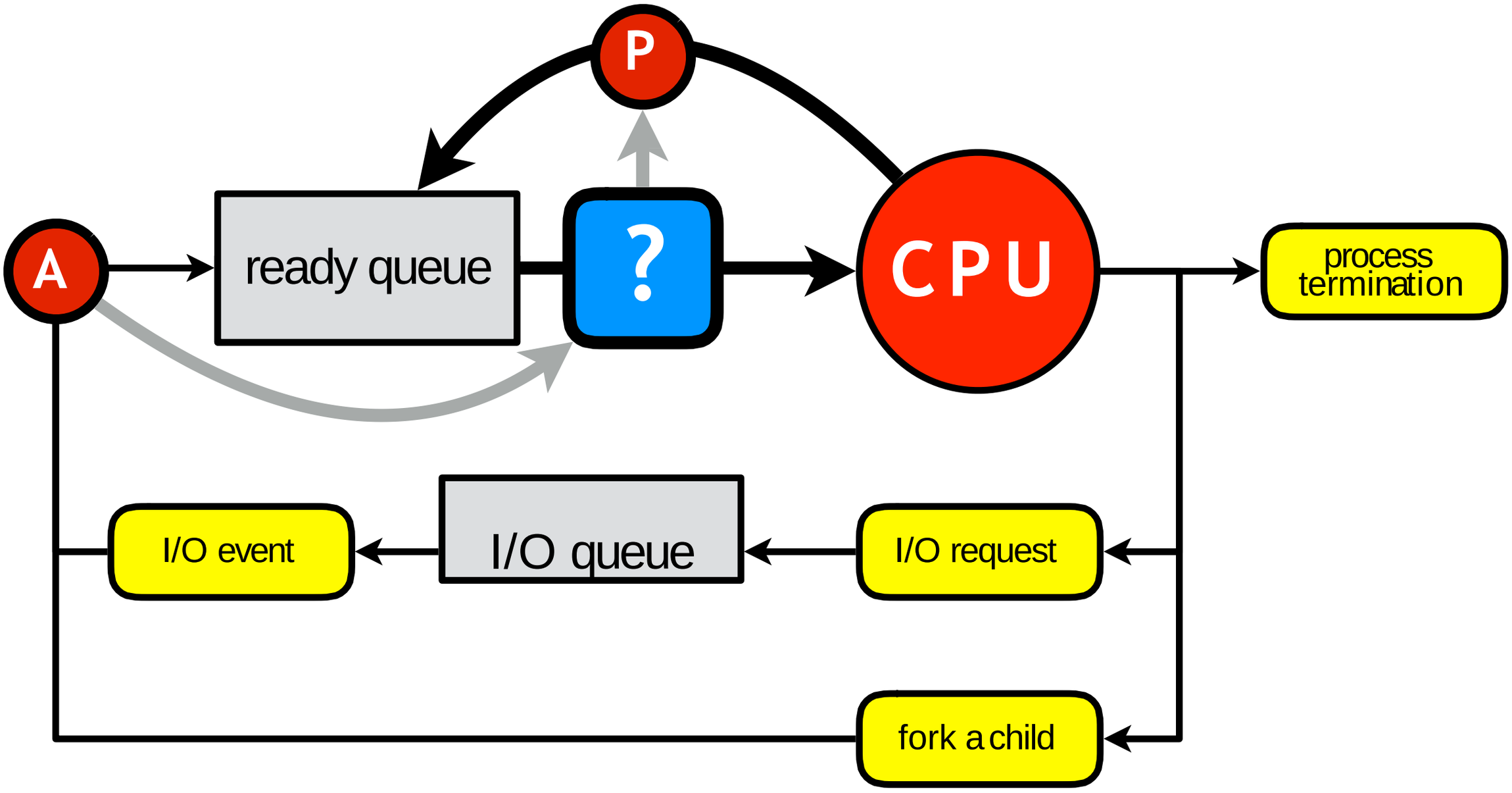
- ★ Also knows as Shortest Process Next (SPN) scheduling.

- ★ Also known as Shortest job next (SJN) scheduling.

- ★ SJF is **optimal** – gives **minimum average waiting** time for a given set of processes.,but we cannot predict the next job CPU burst times dynamically.

A When a process with a shorter burst time compared to the currently scheduled process arrives to the ready queue ...

P ... wouldn't it be more optimal (minimising the average waiting time) to preempt the running process and switch to newly arrived process?

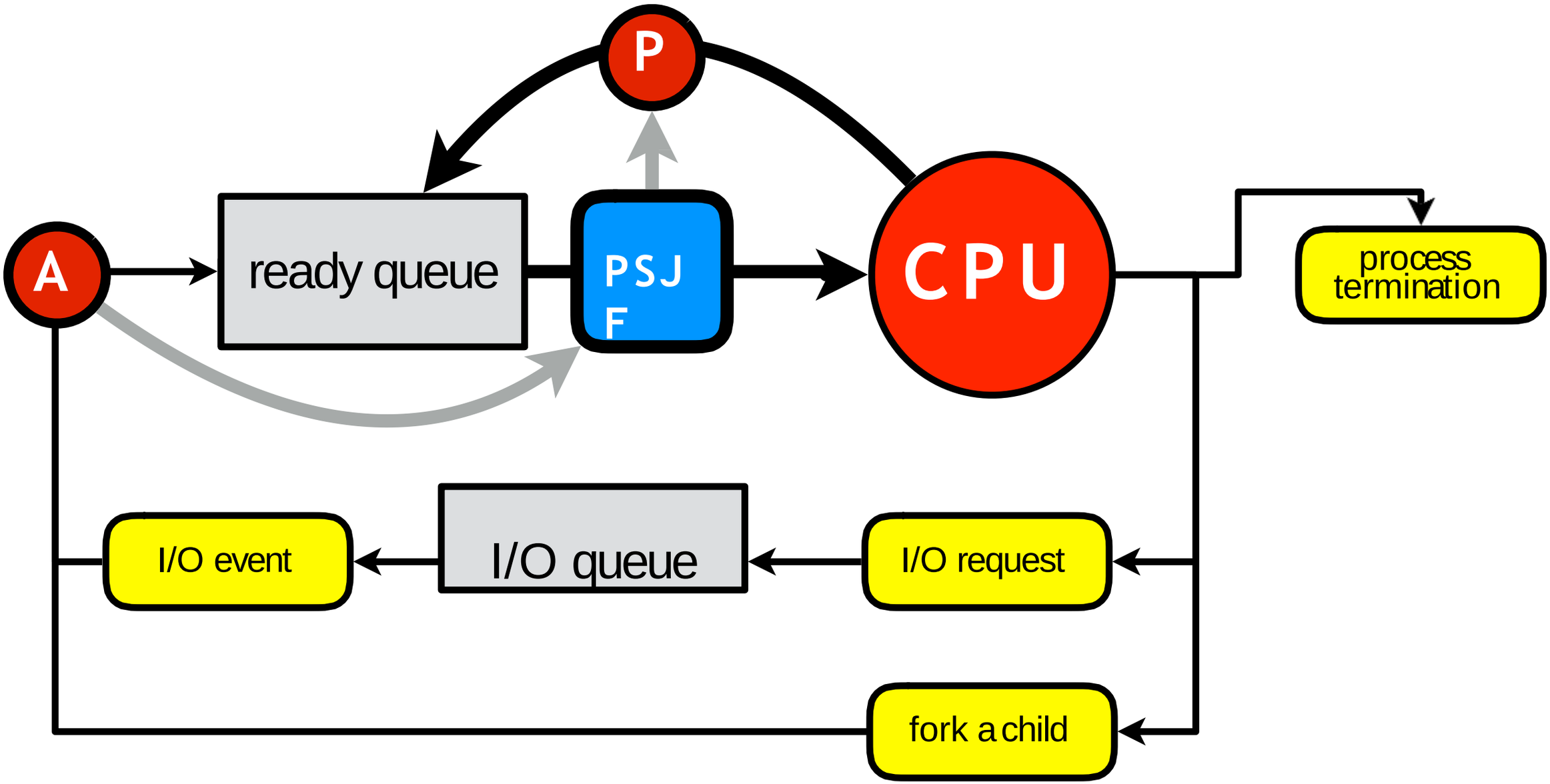


PSJF

Preemptive Shortest Job First

Preemptive Shortest Job First (PSJF)

An extension of SJF where the currently running process is P preempted if the CPU burst of a process A arriving to the ready queue is shorted than the remaining CPU burst of the currently running process.

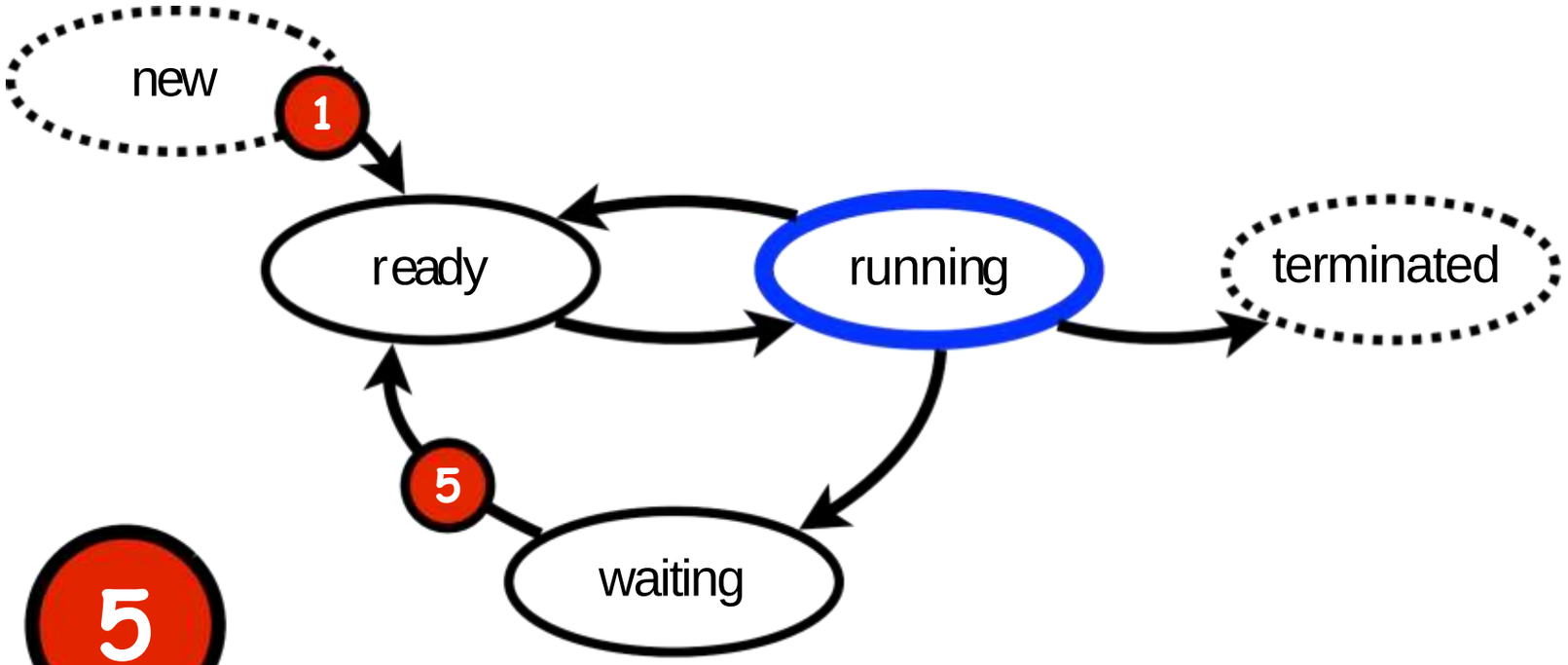


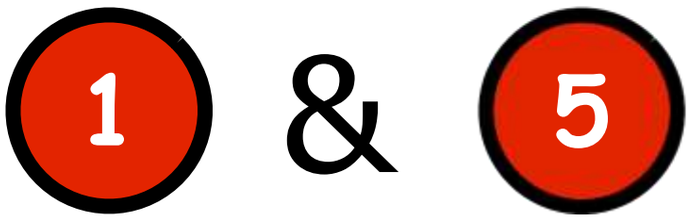
Preemptive Shortest Job First (PSJF)

The currently running process is preempted if the CPU burst of a process arriving to the ready queue is shorted than the remaining CPU burst of the currently running process.

- ★ Also known as shortest remaining time first (SRTF).
- ★ The currently executing process will always run until completion or until a new process is added to the ready queue that requires a smaller amount of time to complete.
- ★ Shortest remaining time is advantageous because **short processes are handled very quickly**.
- ★ Requires very **little overhead** since a decision is made only when a process completes or a new process is added, and when a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently in the ready queue.

SJV vs PSJF





SJF: The currently running process is allowed to continue to execute.

PSJF: The currently running process is preempted if the CPU burst of the newly arrived process is shorter than the remaining CPU burst of the currently running process.

Priority Scheduling

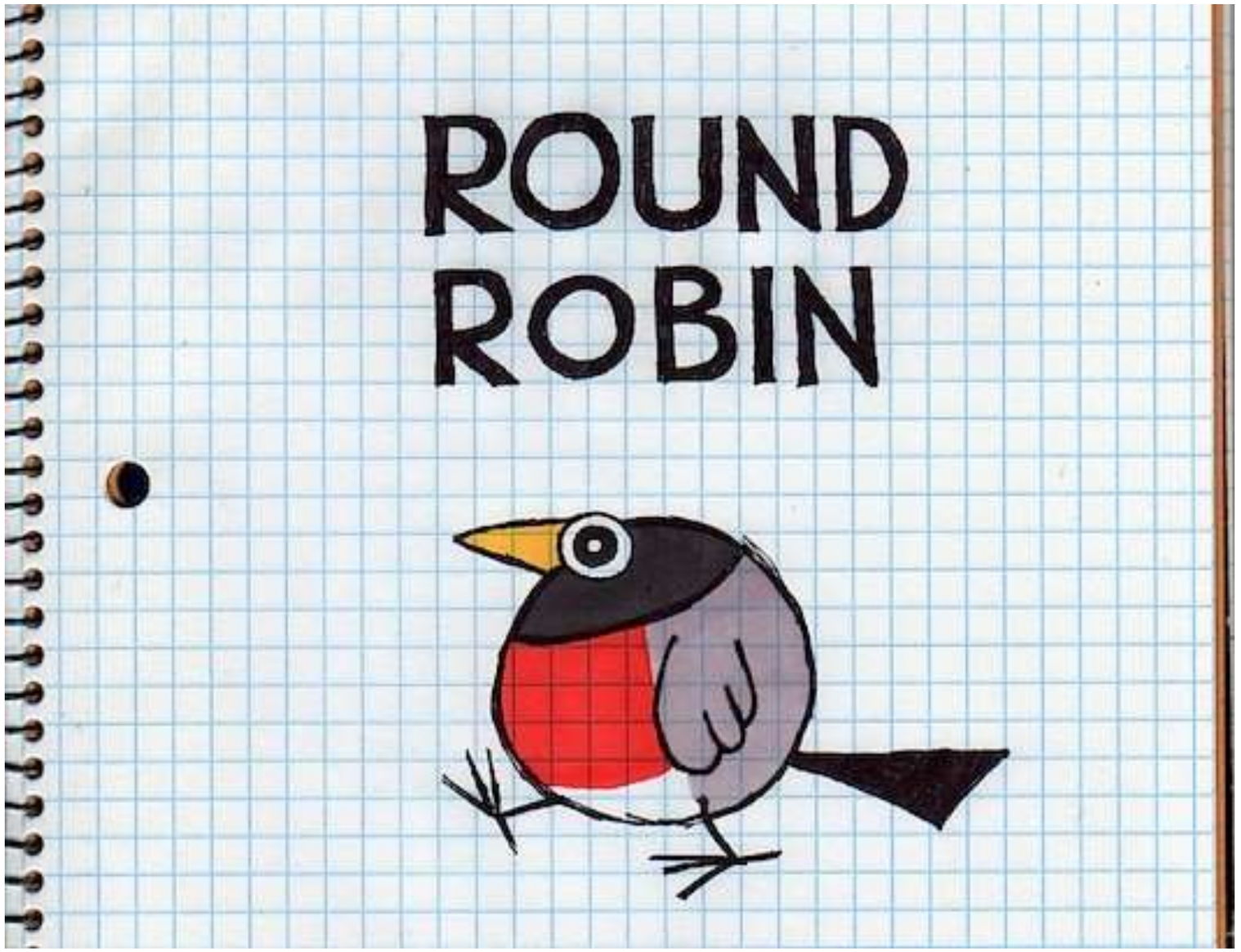
Priority scheduling

A priority number (integer) is associated with each process. The CPU is allocated to the process with the highest priority (smallest integer = highest priority).

- **Preemption** – should a higher priority process be allowed to preempt a running process with lower priority?
- **Starvation** – low priority processes may never execute.
- **Aging** – ensure that jobs with lower priority will eventually complete their execution. Aging can be implemented by increasing the priority of a process as time progresses.

RR

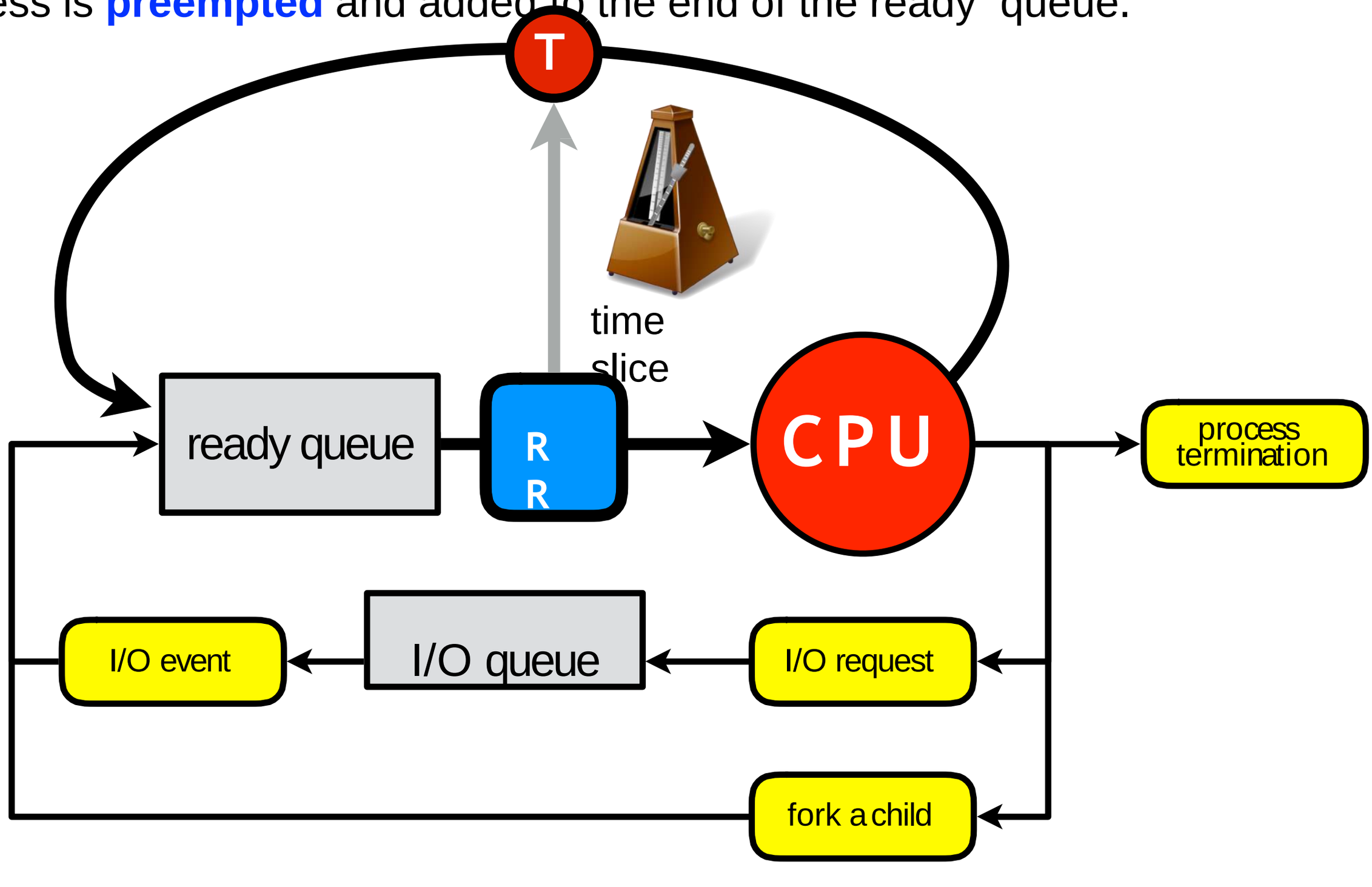
Round Robin



Round Robin is one of the simplest **CPU scheduling** algorithms that also **prevents starvation**.

Round Robin (RR)

Round Robin (RR) is a scheduling algorithm where time slices(usually 10-100 milliseconds) are assigned to each process . After this time has elapsed, the process is **preempted** and added to the end of the ready queue.



Foreground and background processes

Sometimes process can easily be classified into two groups, one set of processes that interacts with users and one set that doesn’t.

Background process (batch)

A process that don’t interacts with any user is called a background process or a batch process.

Foreground process (interactive)

A process that interacts with users is callad a foreground process or an interactive process.

Multilevel queue scheduling

Multilevel queue scheduling

A multi-level queue scheduling algorithm is used in scenarios where the processes can be classified into groups based on properties like process type, CPU time, IO access, memory size, etc

General classification of processes:

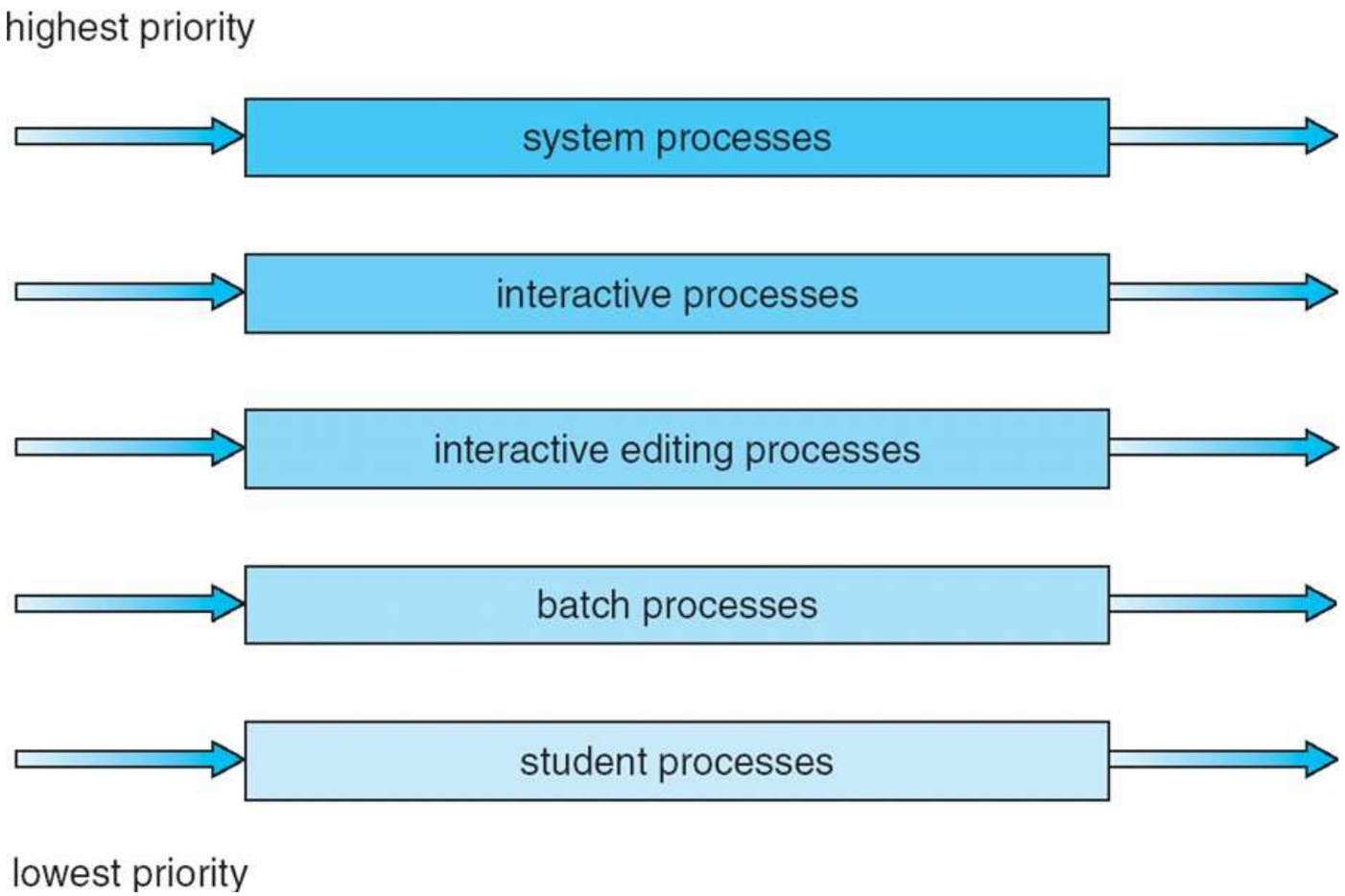
- ★ foreground (interactive)
- ★ background (batch)

In a multi-level queue scheduling algorithm, there will be **n** number of **queues**, where **n** is the number of groups the processes are classified into.

- ★ Each queue will be assigned a priority and will have its own scheduling algorithm like Round-robin scheduling or FCFS.

Multilevel queue scheduling

Use several ready queues. A process is permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.



Each queue has its own scheduling algorithm.

There must be scheduling among the queues.

For example, the foreground queue may have absolute priority over the background queue. If an interactive process enters the ready queue while a batch process is running, the batch process will be preempted.

Multilevel queue scheduling

How to select scheduling algorithms for the various queues?

Ready queue is partitioned into separate queues:

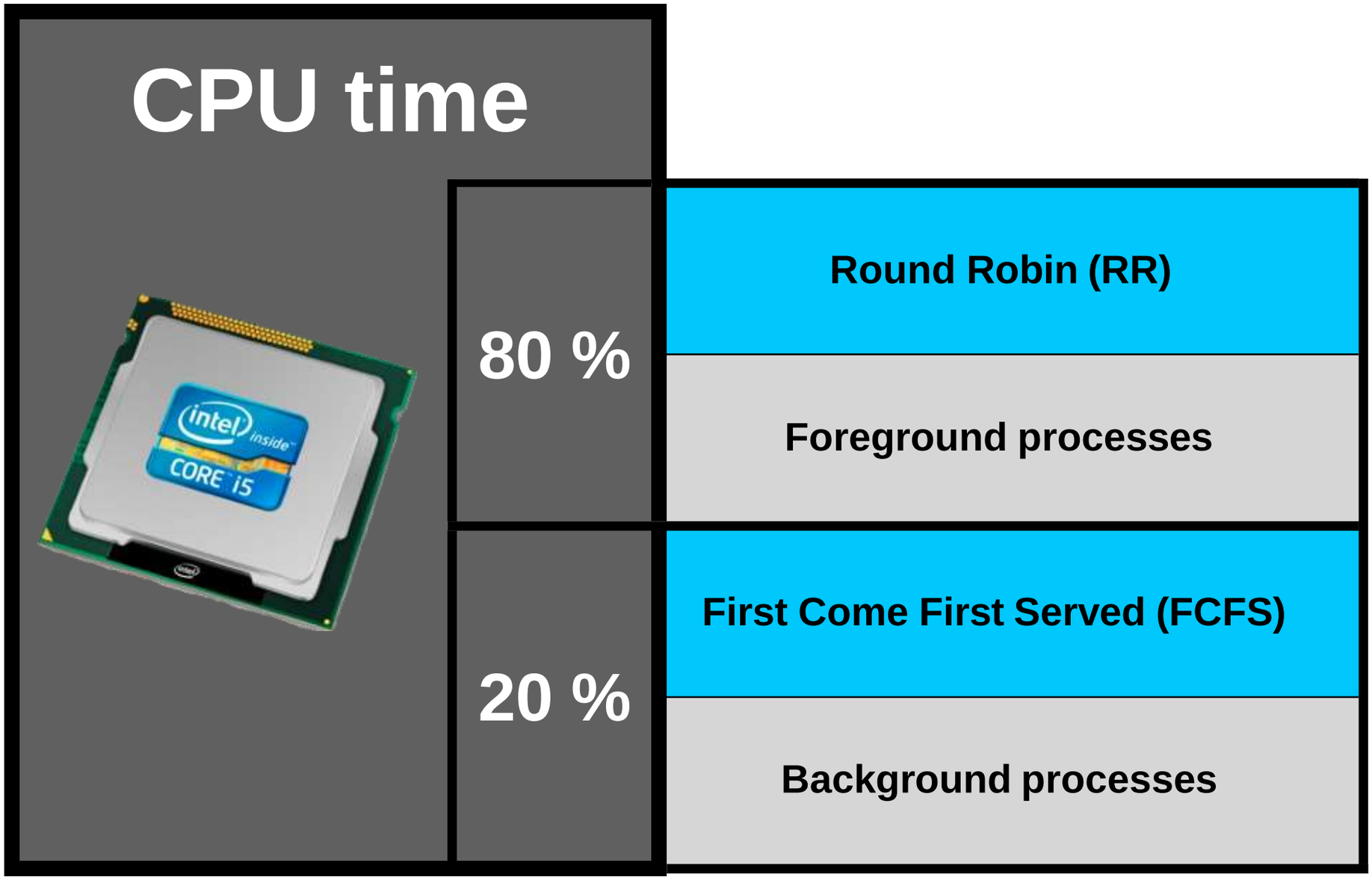
- ★ foreground (interactive)
- ★ background (batch)

Each queue has its own scheduling algorithm

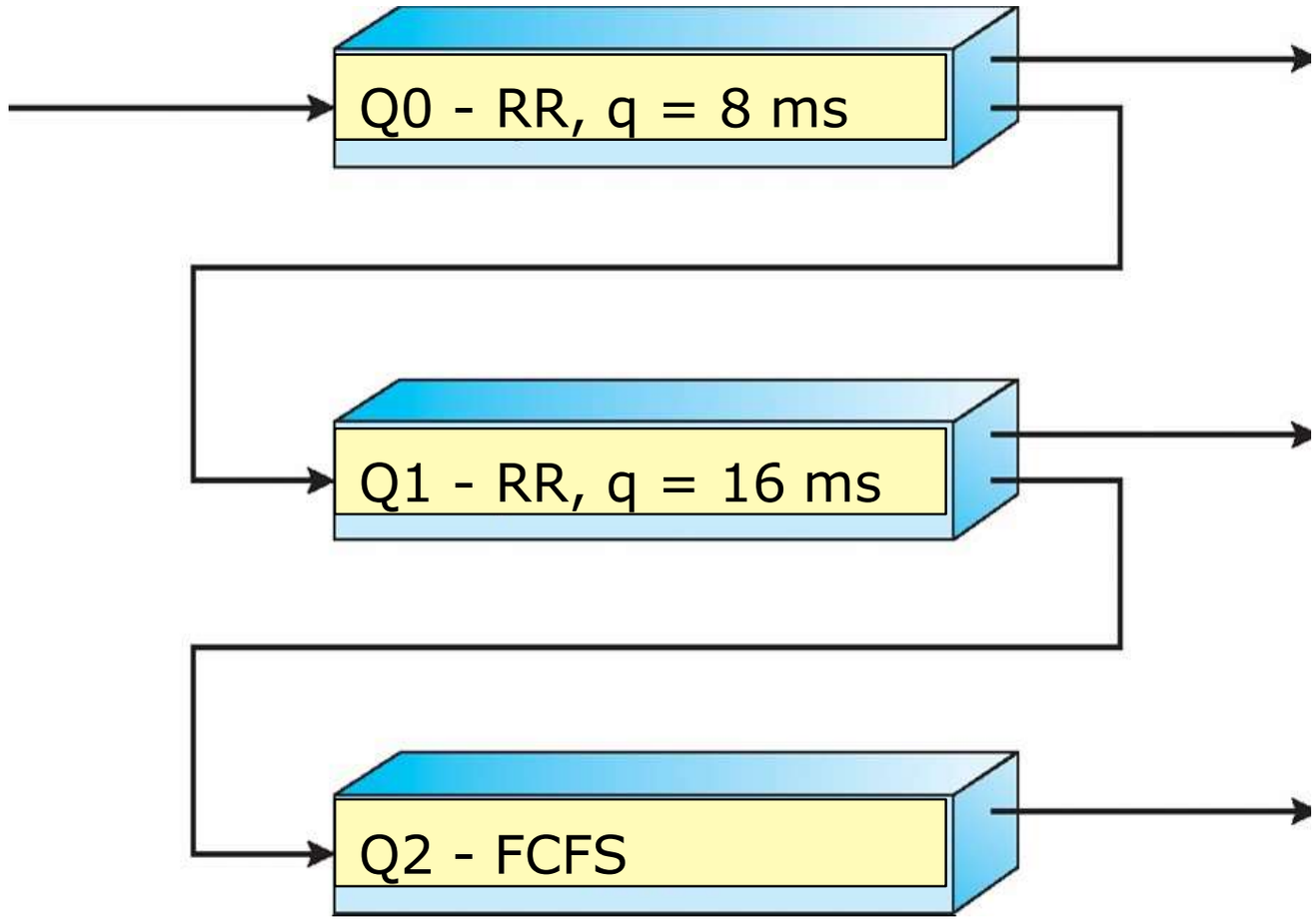
- ★ foreground – RR
- ★ background – FCFS

Multilevel queue scheduling

Example of time slicing among multilevel queues. Foreground process are given 80 % of the CPU time for RR scheduling and background processes are given 20 % of the CPU time for FCFS scheduling.



Multilevel feedback queue scheduling



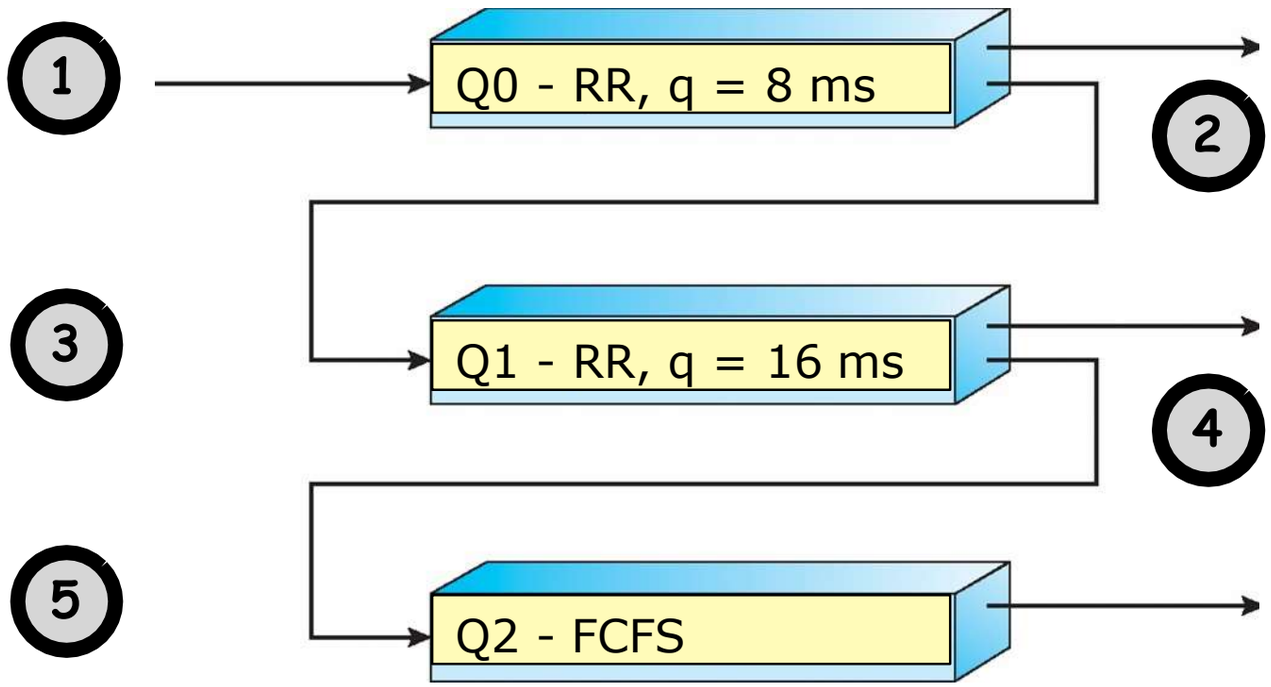
Priorities

- The scheduler first executes all processes in Q0.
- Only when Q0 is empty will it execute processes in Q1.
- Similarly, processes in Q2 will only be executed if Q0 and Q1 are empty.

Preemption

- A process that arrives at Q1 will preempt a process in Q2.
- A process in Q1 will in turn be preempted by a process arriving at Q0.

Example

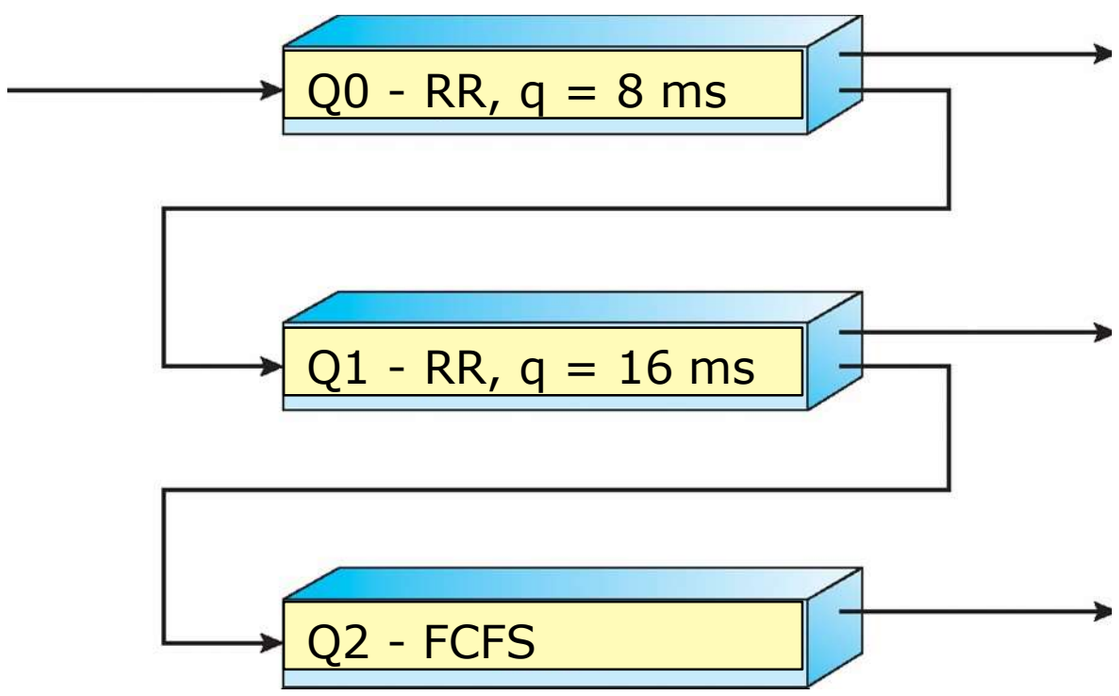


1. A new job enters queue Q0 which is served RR. Each job gets at most 8 milliseconds of CPU time.
2. If it does not finish in 8 milliseconds, the job is moved to queue Q1.
3. At Q1 the job is again served RR and receives 16 additional milliseconds.
4. If it still does not complete, it is preempted and moved to queue Q2.
5. Once in Q2, processes are scheduled using FCFS but are run only when Q0 and Q1 are empty.

Aging

Aging is used to ensure that jobs with lower priority will eventually complete their execution.

- ★ Aging can be implemented by increasing the priority of a process as time progresses.



Multilevel feedback queue scheduling

A process can move between the various queues.

- ★ Aging can be implemented this way.

Multiprocessing

Module 3

Operating systems 2018

1DT044 and 1DT096

February 2018 karl.marklund@it.uu.se Uppsala University

Multiprocessing (1)

Multiprocessing is the use of more than one CPU in a computer system.

★ The CPU is the arithmetic and logic engine that **executes user applications**.

★ With multiple CPUs, **more than one** set of **program** instructions can be **executed at the same time**.

★ All of the CPUs have the same user-mode instruction set.

★ A running job can be rescheduled from one CPU to another.

Source: https://en.wikipedia.org/wiki/Multiprocessing 2016-02-04

https://en.wikipedia.org/wiki/Asymmetric_multiprocessing 2016-02-04

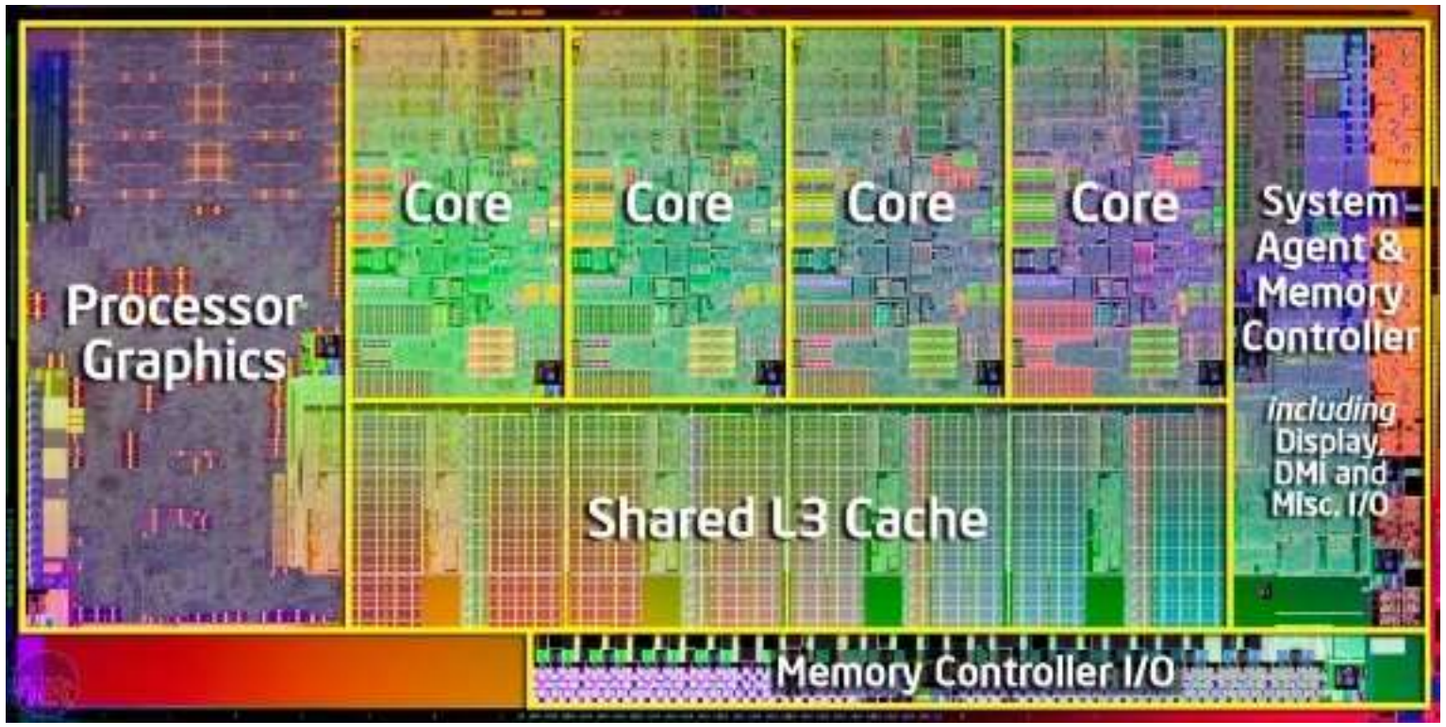
Multiprocessing (2)

The definition of multiprocessing can vary with context, mostly as a functionof how CPUs are defined.

- Multiple cores on one die
- Multiple dies in one package
- Multiple packages in one system unit

Multicore

A multi-core processor is a single computing component with two or more independent actual processing units (aka cores), which are the units that read and execute program instructions.



A quad core processor with a single GPU and a single shared L3 cache.

Asymmetric multiprocessing (AMP)

In an asymmetric multiprocessing system, not all CPUs are treated equally.

For example, a system might only allow one CPU:

- ★ to execute operating system code and accesses the operating system data structures, alleviating the need for data sharing
- ★ to perform I/O operations.

Other AMP systems:

- ★ could allow any CPU to execute operating system code and perform I/O operations, so that they were symmetric with regard to processor roles
- ★ but attached some or all peripherals to particular CPUs, so that they were asymmetric with regard to peripheral attachment.

Symmetric multiprocessing (SMP)

A symmetric multiprocessor system (SMP) is a multiprocessor system with centralized shared memory called main memory (MM) operating under a single operating system with two or more homogeneous processors.

Each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes.

Processor affinity **(2)**

The binding and unbinding of a process or a thread to a central processing unit (CPU) or a range of CPUs.

Soft affinity

- The tendency of a scheduler to try to keep processes on the same CPU as long as possible. It is merely an attempt; if it is ever infeasible, the processes certainly will migrate to another processor.

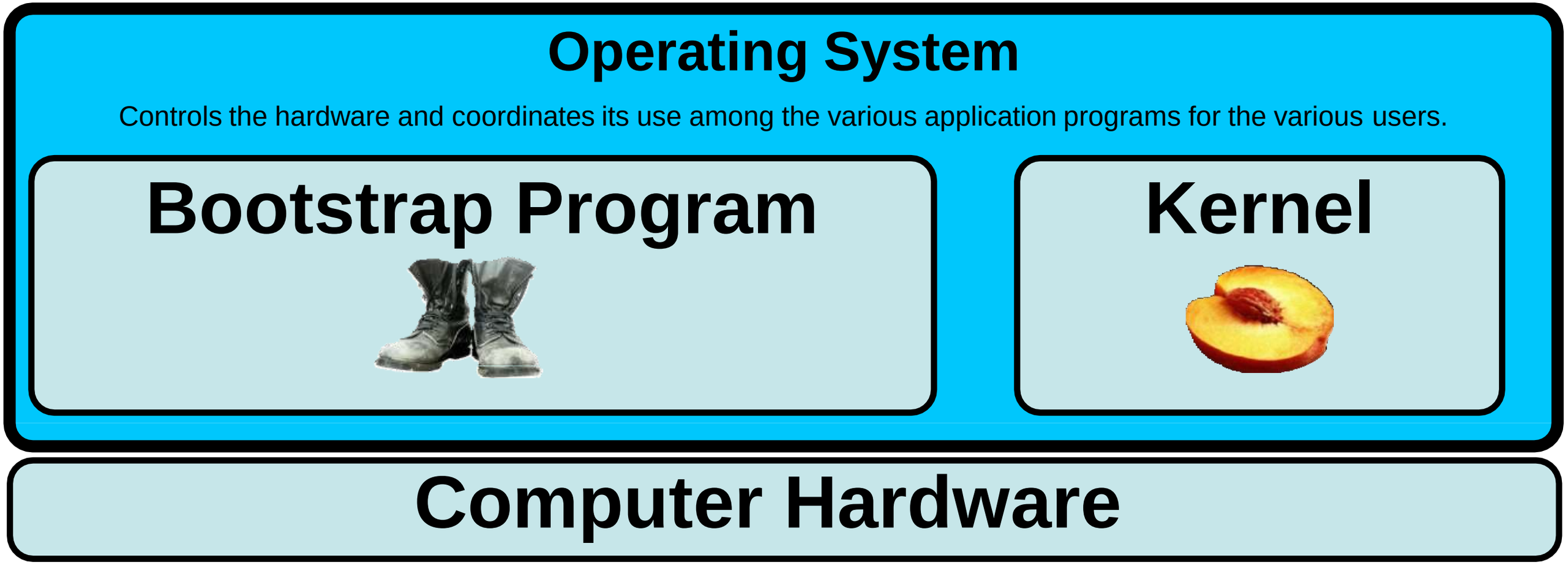
Hard affinity

- It is a requirement, and processes must adhere to a specified hard affinity. If a processor is bound to CPU zero, for example, then it can run only on CPU zero. At the time of resource allocation, each task is allocated to its kin processor in preference to others.

System Boot

The operating system must be made available to hardware so hardware can start it:

- Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
- Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
- When power initialized on system, execution starts at a fixed memory location
- **Firmware** (ROM/EEPROM) used to hold initial boot code.



System and Application Programs

Operating System

Controls the hardware and coordinates its use among the various application programs for the various users.

Bootstrap program

Kept on chip (ROM or EEPROM), aka **firmware**.

Small program executed on power up or reboot.

Initializes all aspects of the system, from CPU register to device controllers to memory content.

Locates and **loads the kernel into memory** for execution.

Kernel

The part of the operating system that is running at all times.

On boot, starts executing the first process such as **init**.

Waits for some **event** to occur ...

Computer Hardware

IPC- Inter Process Communication

How can we make processes communicate, sharing information?

Message-Passing

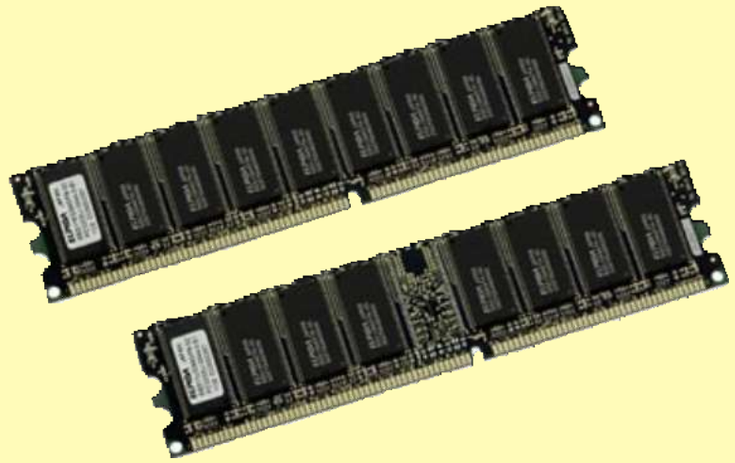
Messages can be exchanged between processes either directly or indirectly using a common mailbox.

The PID of the receiver is passed to the general-purpose **`open`** and **`close`** system calls (file system) or to open connection and close connection system calls (networking), depending on the systems's model of communication.

The recipient process usually must give its permission for communication to take place with an accept connection system call.

Shared-Memory

Processes can communicate by reading and writning to shared memory.



Processes uses **`shared memory create`** and **`shared memory attach`** system calls to create and gain access to regions of memory owned by other processes.

Normally, the OS tries to prevent one process from accessing another process's memory.

Shared-memory requires that two or more processes agree to remove this restriction.

Message passing vs Shared memory

Message-Passing

Messages can be exchanged between processes either directly or indirectly using a common mailbox.

Shared-Memory

Processes can communicate by reading and writning to shared memory.

Pros & Cons of the two approaches?

Message-passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.

It is also easier to implement compared to the shared memory approach.

Shared memory allows maximum speed and convenience of communication, since it can be done at memory transfer speeds.

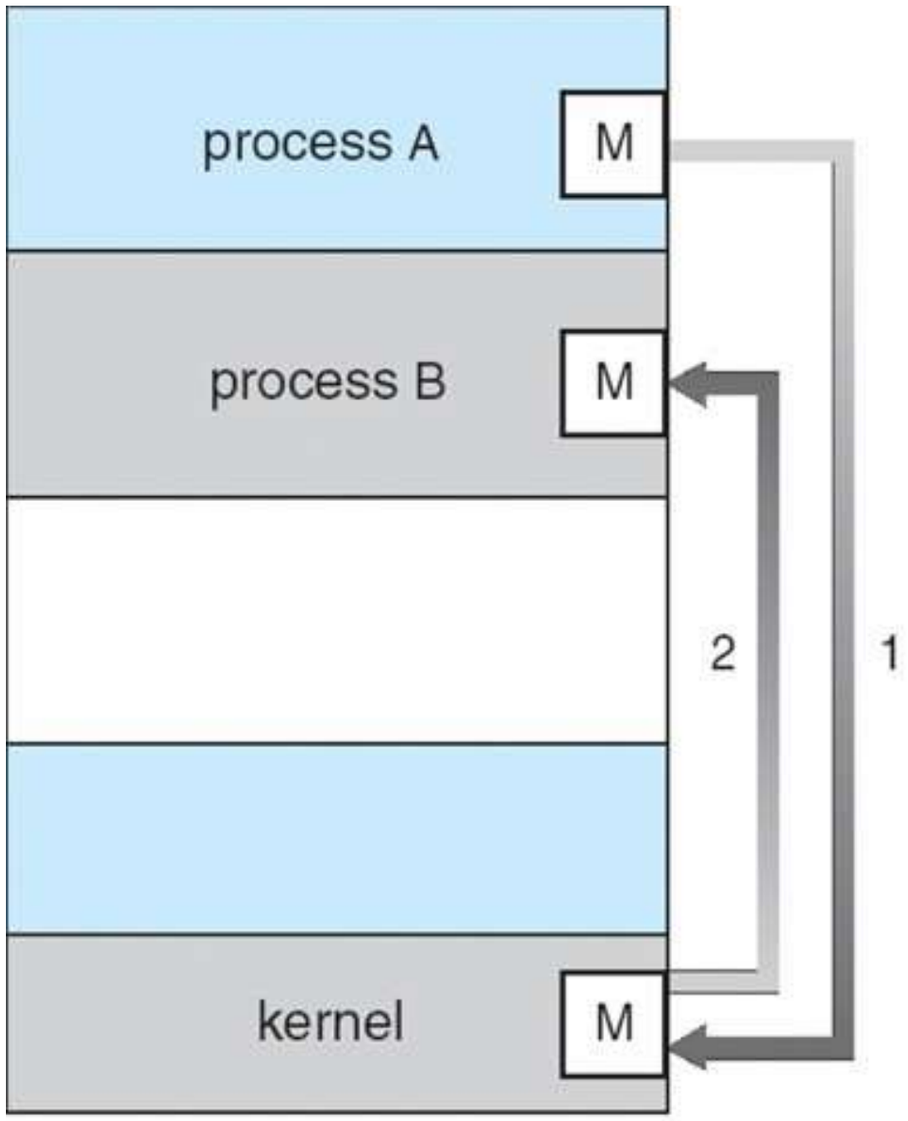
Problems: protection and synchronization between the processes sharing memory.

Inter process communication (IPC)

How can we make processes communicate (sharing information)?

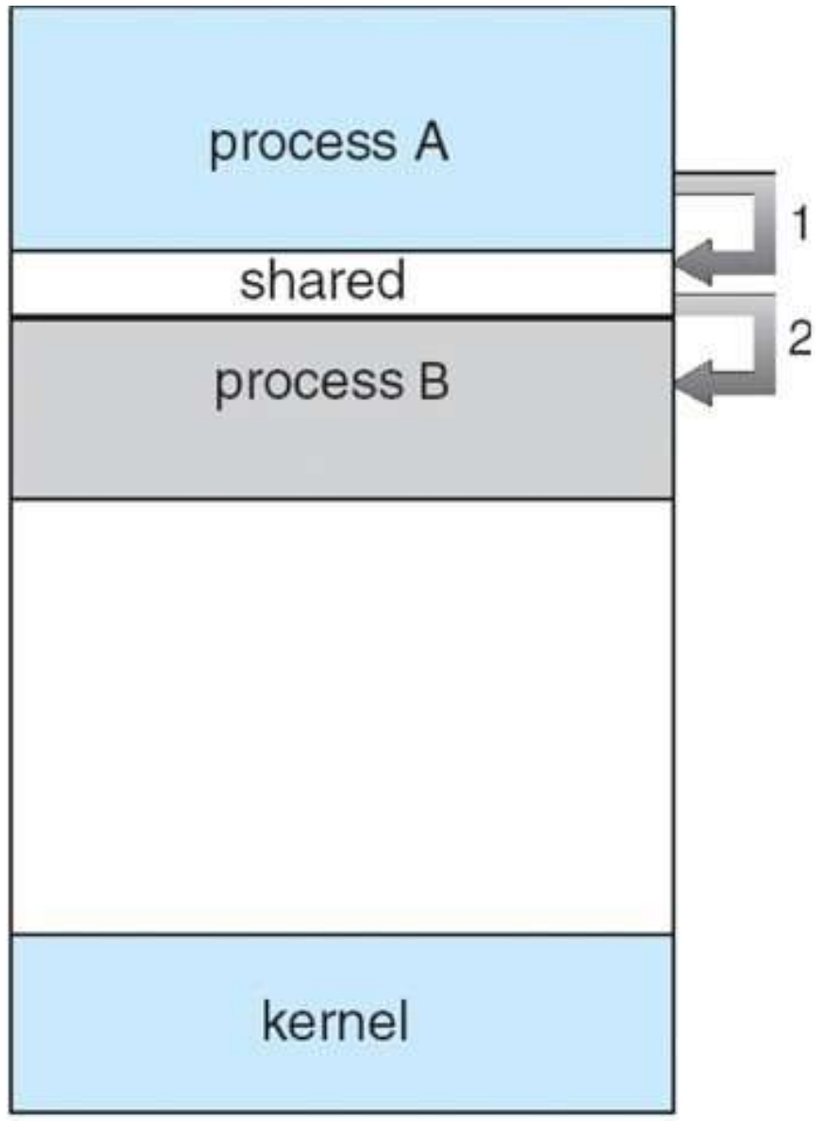
Two methods: **message passing** and **shared memory**.

Message Passing



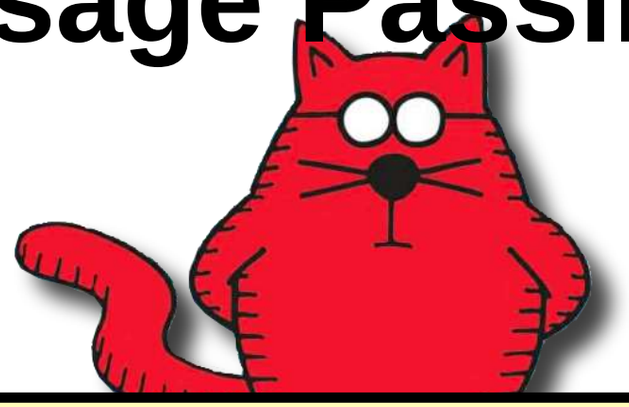
Communication take place by means of messages exchanged between the cooperating processes.

Shared Memory



A region of memory that is shared by cooperating process is established. Process can then exchange information by reading and writing data to the shared region.

Message Passing



Messages can be exchanged between processes either directly or indirectly using a common mailbox.

The recipient process usually must give its permission for communication to take place with an accept connection system call.

Message-passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.

Easier to implement compared to the shared memory approach.

Shared Memory

Processes can communicate by reading and writing to shared memory.

Normally, the OS tries to prevent one process from accessing another process's memory. Shared-memory requires that two or more processes agree to remove this restriction.

Shared memory **allows maximum speed** and convenience of communication, since it can be done at memory transfer speeds.

Problems: protection and synchronization between the processes sharing memory.

Pipes

Pipes

An (anonymous) pipe is a **simplex FIFO communication channel** that may be used for one-way interprocess communication (IPC).

★ An implementation is often integrated into the operating system's file IO subsystem.

★ Pipes where one of the first IPC mechanisms in early UNIX systems.

★ Typically a parent program opens anonymous pipes, and creates a new process that inherits the other ends of the pipes, or creates several new processes and arranges them in a pipeline.

Producer and consumer

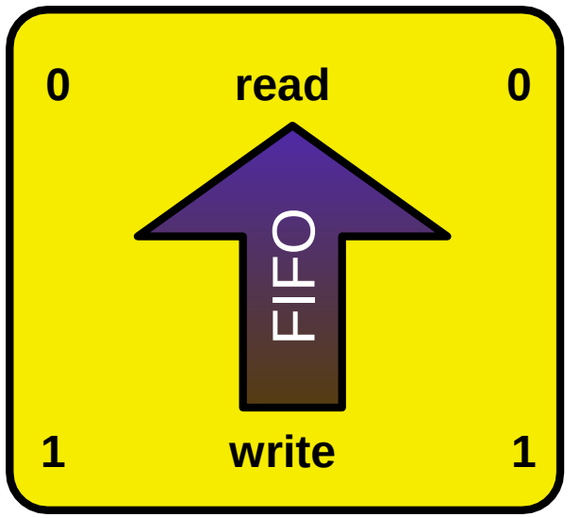
Anonymous pipes allow two processes to communicate in standard producer consumer fashion.

- ★ The **producer** writes to one end of the pipe (the write end).
- ★ The **consumer** reads from one end of the pipe (the read end)
- ★ As a result, ordinary pipes are **unidirectional**, allowing only **one-way communication**.
- ★ If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.

pipe()

In Unix-like operating systems pipes are created using the **`pipe()` system call**.

- ★ As a result of the pipe() system call, a pipe object is created.
- ★ A pipe is a FIFO buffer that can be used for inter process communication (IPC).
- ★ The pipe() system call **returns a pair of file descriptors** referring to the **read** and **write** ends of the pipe.



Using pipe()

User Space

Kernel Space

Parent Process

```
int pfd[2];
pipe(pfd);
// pfd[0] = 3

// pfd[1] = 4
```

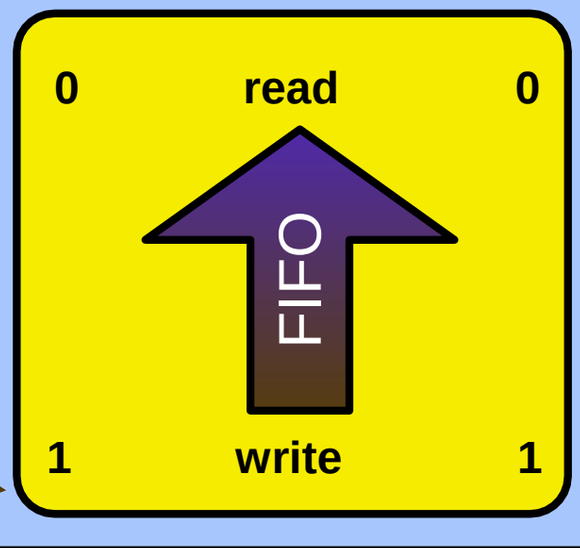
Descriptors	
0	stdin
1	stdout
2	stderr
3	pipe read
4	pipe write

(1)

As a result of the **pipe()** system call, a pipe object is created. A pipe is a FIFO buffer that can be used for inter process communication (IPC).

(2)

As a result of the **pipe()** system call, the elements of the array pfd are assigned descriptor numbers to the created pipe.



Now a single process can write data to, and later read data from the pipe ...

Pipe **and** **For k**

Using `pipe()` together with `fork()`

User Space

Kernel Space

Parent Process

```
int pfd[2];
pipe(pfd);
// pfd[0] = 3
// pfd[1] = 4
fork();
```

fork()

Child Process

After **fork()**, the child process have the same set of descriptors including read and write descriptors to the pipe.

Descriptors	
0	stdin
1	stdout
2	stderr
3	pipe read
4	pipe write

0 read 0

FIFO

1 write 1

Descriptors	
0	stdin
1	stdout
2	stderr
3	pipe read
4	pipe write

Using `pipe()` together with `fork()`

User Space

Kernel Space

Parent Process

```
int pfd[2];
pipe(pfd);
// pfd[0] = 3
// pfd[1] = 4
fork();
close(pfd[0]);
```

The parent can close the read descriptor to the pipe.

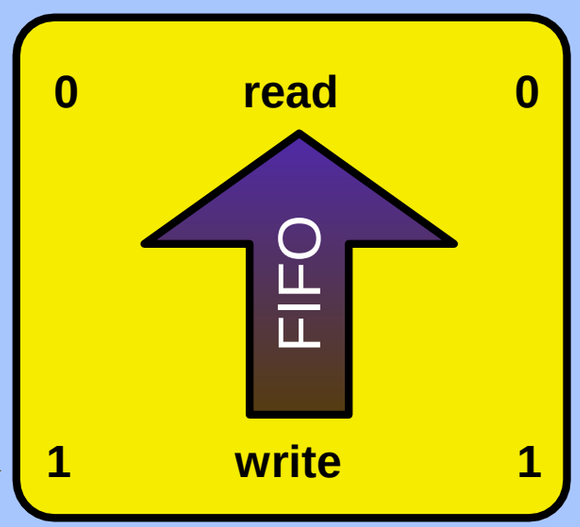
Now, the parent can act as a single producer and the child as a single consumer of data through the pipe using the **read()** and **write()** system calls - just as if it was a file.

Child Process

The child can close the write descriptor to the pipe.

```
close(pfd[1]);
```

Descriptors	
0	stdin
1	stdout
2	stderr
3	~~pipe read~~
4	pipe write



Descriptors	
0	stdin
1	stdout
2	stderr
3	pipe read
4	~~pipe write~~

Blocking

Reading and writing to a pipe may block a process.

Attempt	Conditions	Result
Read	Empty pipe, writer attached	Reader blocked
Write	Full pipe, reader attached	Writer blocked
Read	Empty pipe, no writer attached	EOF returned
Write	No reader	?

Pipes and signals

SIGPIPE

The SIGPIPE signal is used to notify a producer (writer) when there is no longer a consumer (reader) attached to a pipe.

Attempt	Conditions	Result
Read	Empty pipe, writer attached	Reader blocked
Write	Full pipe, reader attached	Writer blocked
Read	Empty pipe, no writer attached	EOF returned
Write	No readers attached	SIGPIPE

By default, SIGPIPE causes the process to terminate. Don't forget to close unused pipe file descriptors using the `close()` system call.