# The process concept and inter process communication

# Lecture 2

## Module 2

---

## Operating systems 2018

### 1DT044 and 1DT096

User 1 User 2 User 3 User N

Compiler Spreadsheet Text Editor Pacman

**System and Application Programs**

# Operating System

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

**Computer Hardware**

# Multiprogramming

## Job

In a multiprogramming system a program loaded to memory and ready to execute is called a job.

## Execute another job while waiting for I/O

The simple idea is to make a job wait for I/O "outside" the CPU. When a job makes a request for I/O the CPU will execute another job while the the first job waits for the/O request to complete.

## Interrupts

Interrupts are used to notify the system when an I/O request is completed.

# States

In a multiprogramming system, a job can be in one of three states.

## Running

The job is currently executing on the CPU. At any time, at most one job can be in this state.

## Ready

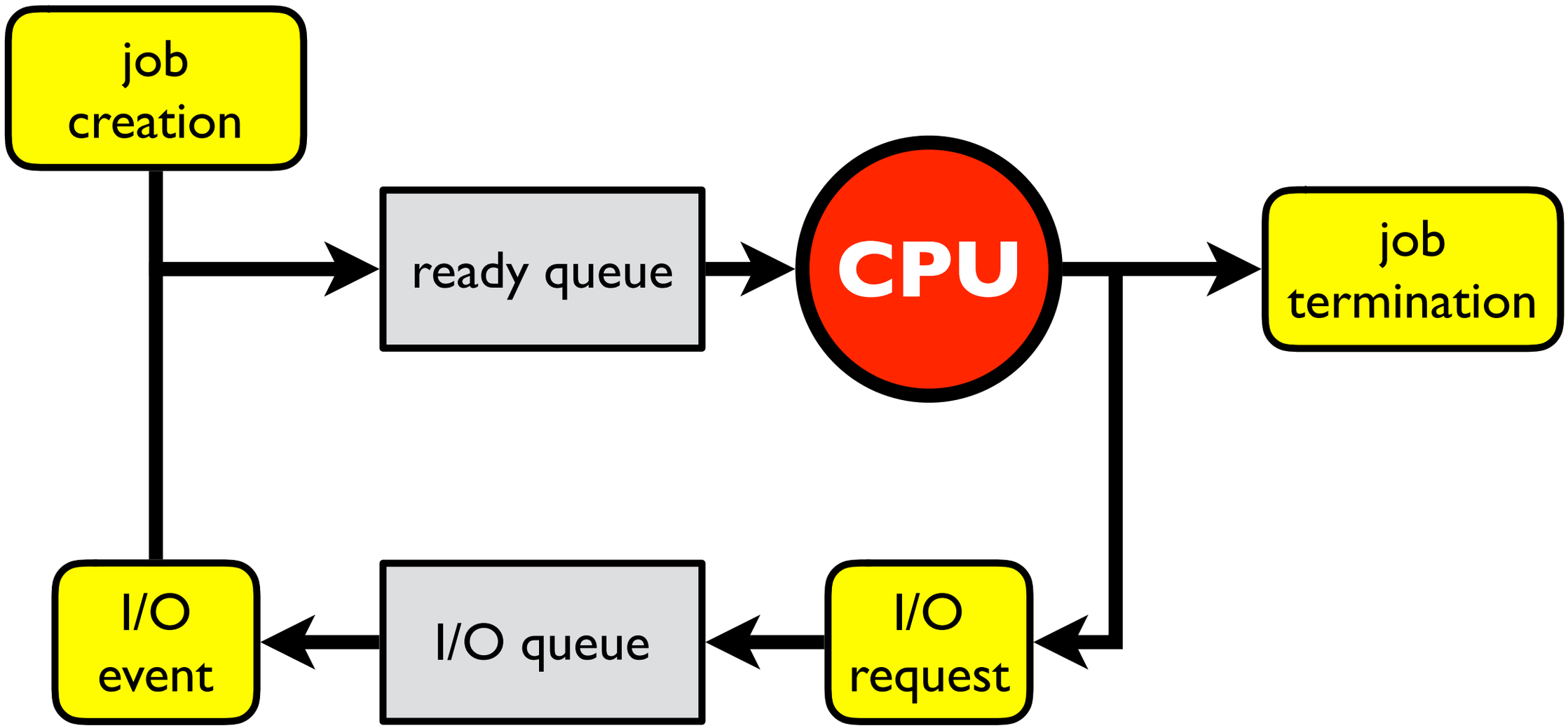
The job is ready to run but currently not selected to do so.

## Waiting

The job is blocked from running on the CPU while waiting for an I/O request to be completed.

# Multiprogramming

A schematic view of multiprogramming



# Observations

Multiprogramming keeps the CPU busy despite individual jobs waiting for I/O.

This seems good if we want to maximize the CPU utilization.

# Problems

Each job might execute quite some time before any other job gets a spin on the CPU.

We must ensure that the operating system maintains control over the CPU.

We cannot allow a user program to get stuck in an infinite loop or never to request I/O.

# Solution

Use a timer. The timer can be set to interrupt the currently executing job after a specified period of time if the job does not request I/O.

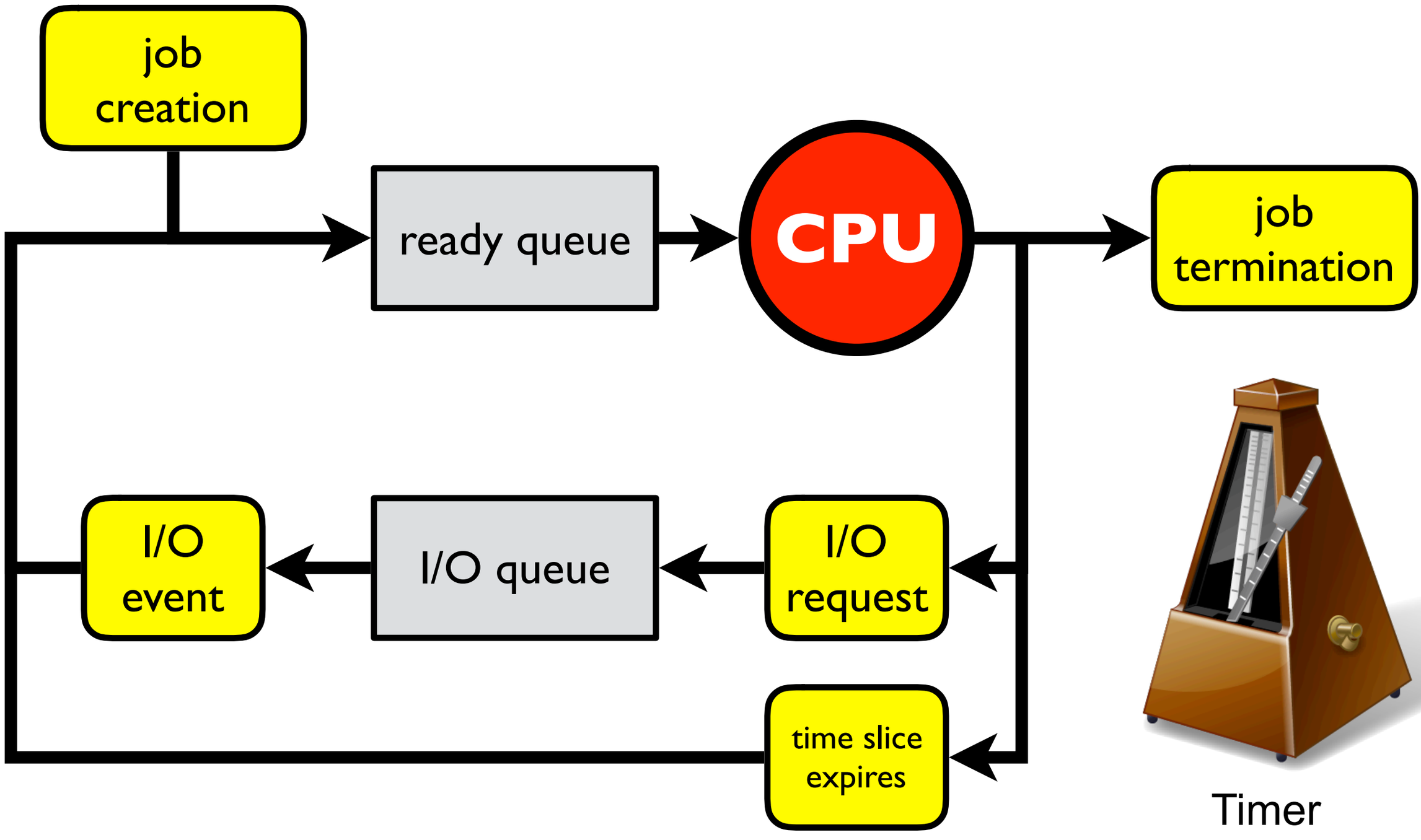
# Multitasking

Multitasking (aka **time sharing**) is a logical extension of multiprogramming where a **timer** is set to cause an interrupt at a regular time interval.

The running job is replaced if the job requests I/O or if the job is interrupted by the timer. This way, the running job is given a **time slice** of execution than cannot be exceeded.

# Multitasking

A schematic view of multitasking






Timer

# Design objectives

Multiprogramming and time sharing are designed to achieve different goals.

- ★ The objective of **multiprogramming** is to have some job running at all time, to **maximize CPU utilization**.
- ★ The objective of **time sharing** is **user interaction**, to switch the CPU among jobs so frequently that users can interact with each program while they are running.

# Important definitions

Input devices


CPU

Control unit

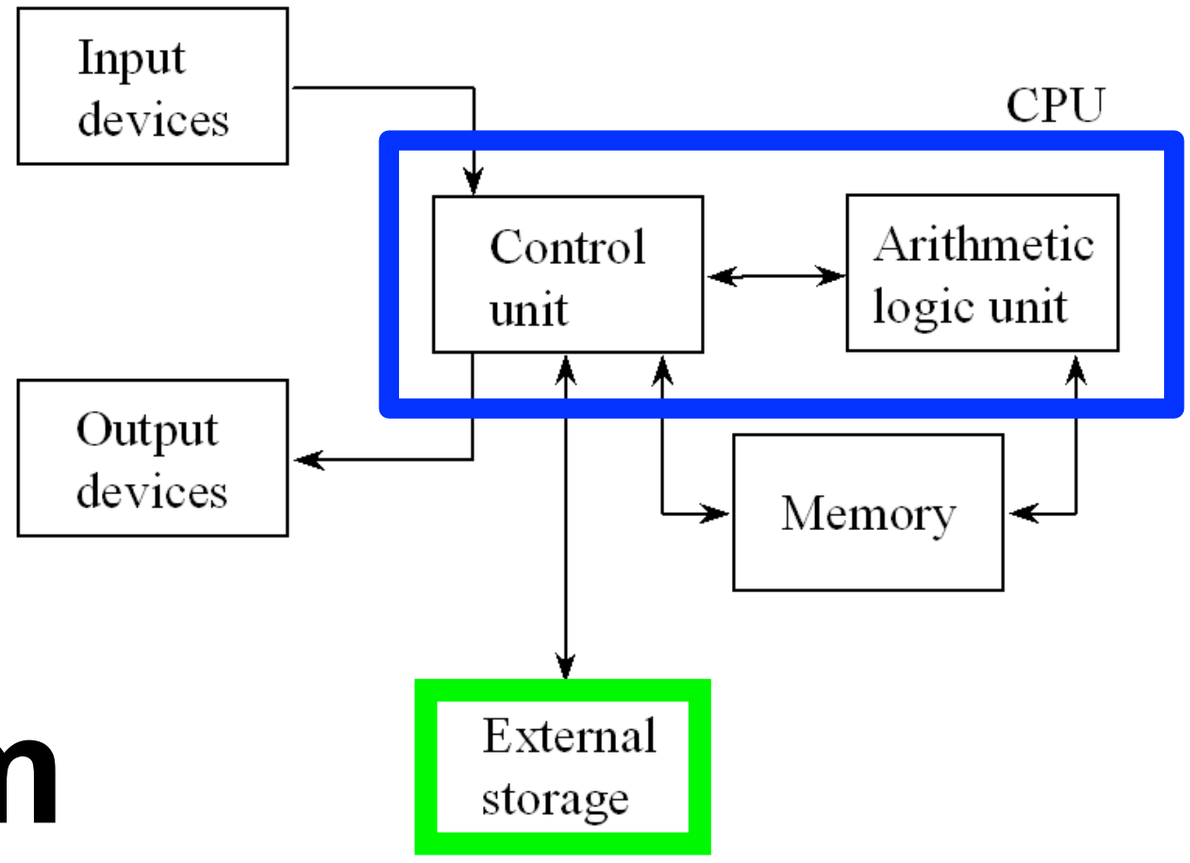
Arithmetic logic unit

Output devices

Memory

External storage

# The von Neuman modell

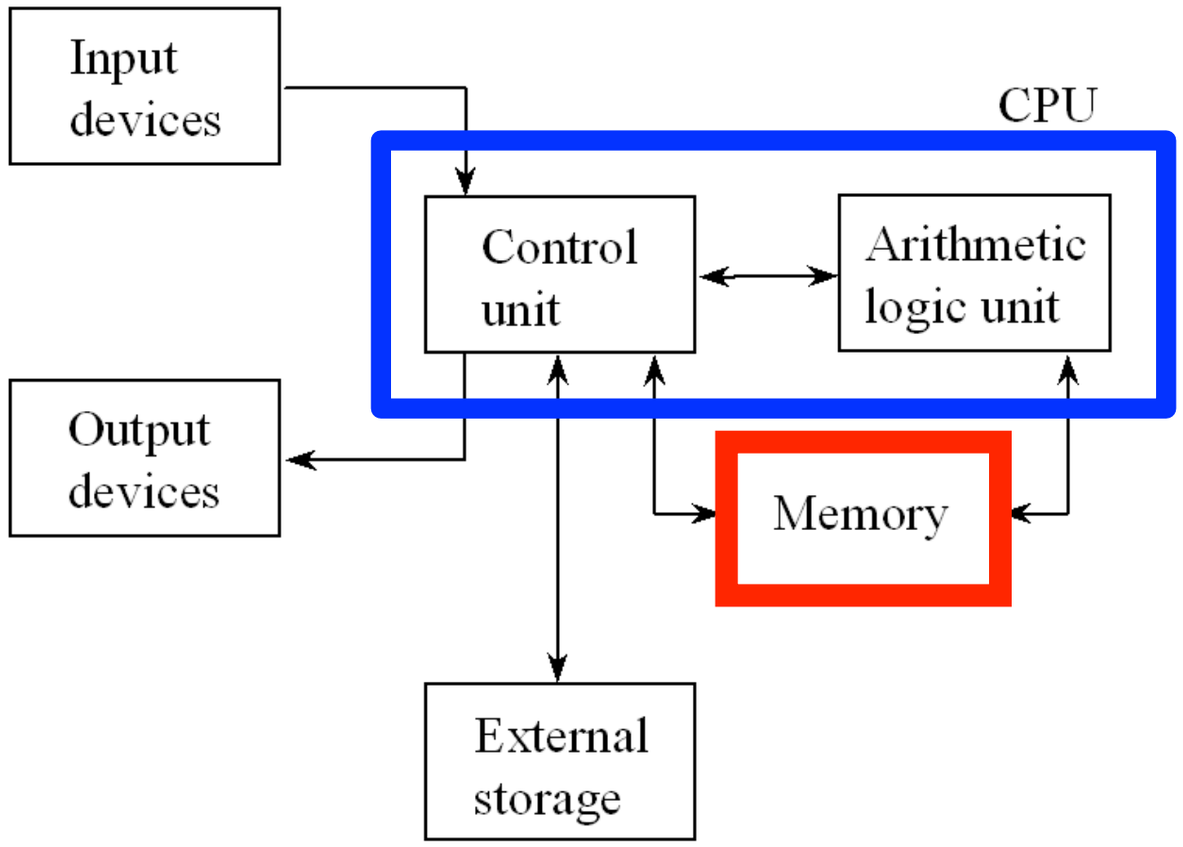



# Program

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).

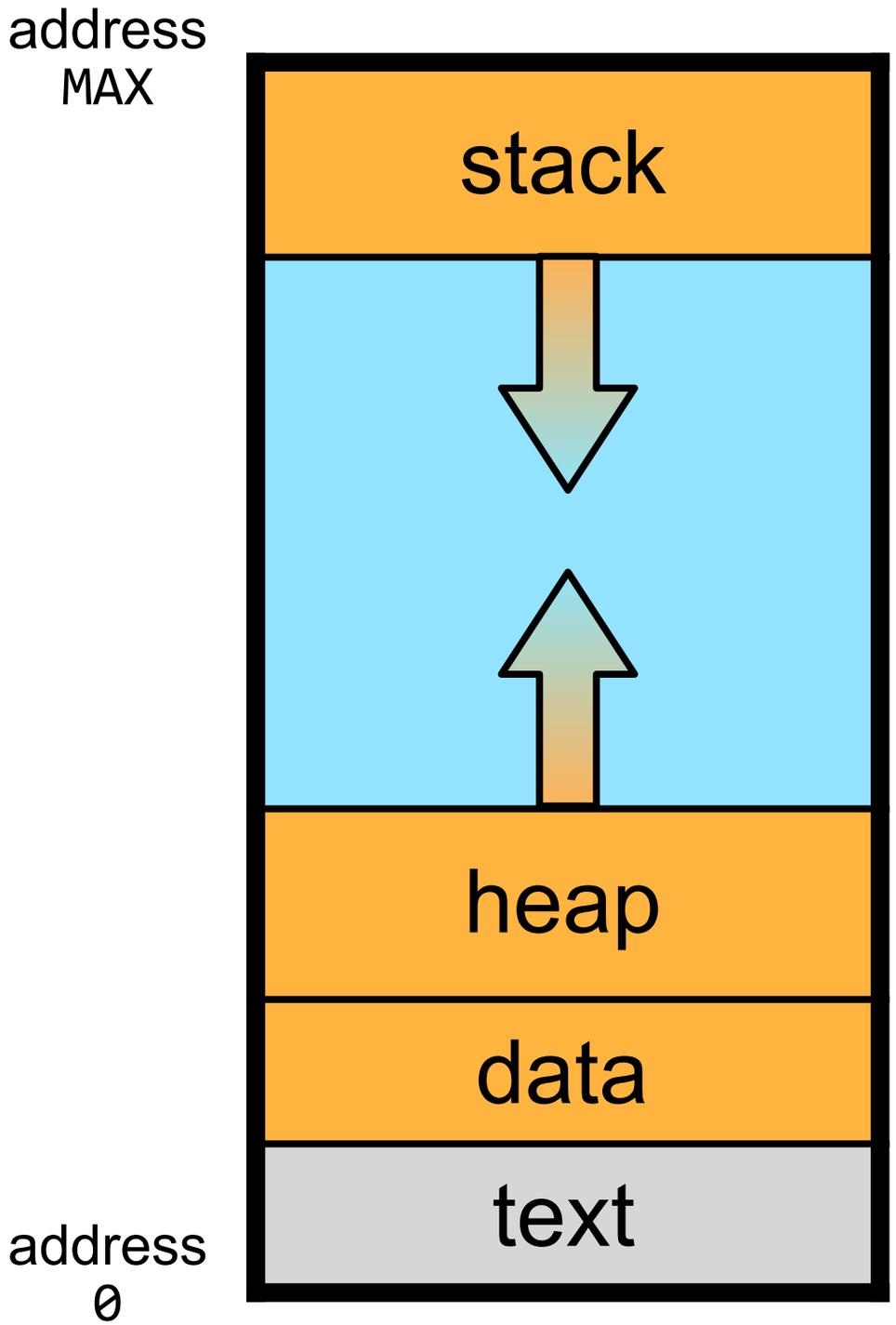



# Process

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.

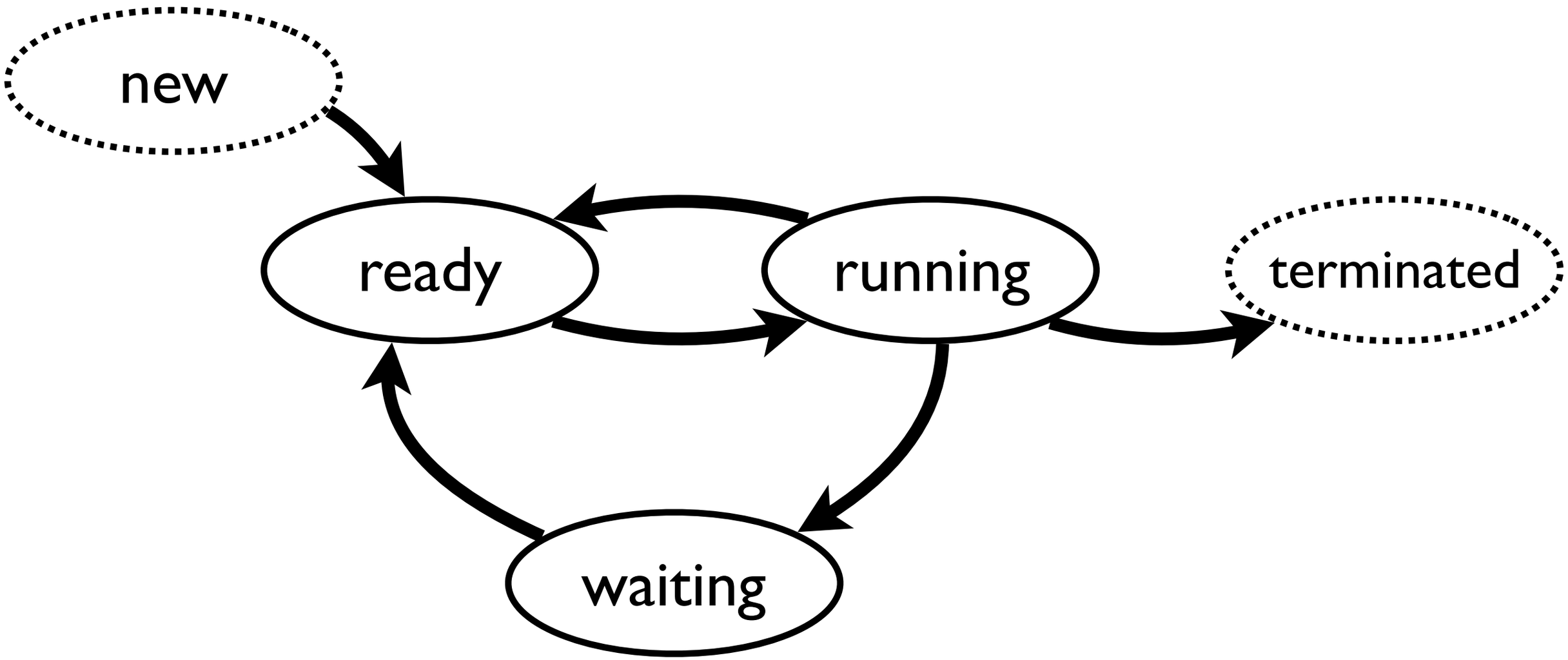
The heap grows from low addresses towards higher addresses.

**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.



# Example program

example.c

```c
// Example C program

#include <stdlib.h>


int x;

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

In this example we start with a single file **`example.c`** file.

```
$ ls

example.c
```

**`file`** is a standard Unix/Linux program for recognizing the type of data contained in a computer file.

```
$ file example.c

example.c: C source, ASCII text
```

Here we see that **`example.c`** is an ordinary C source file.

# 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
```

The compiler **`gcc`** transformed the **`example.c`** program into the **executable** **`a.out`**.

```
$ file a.out

a.out: ELF 64-bit LSB executable, x86-64,
version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 2.6.32,
BuildID[sha1]=4a384d17163f5f8c65cc65b87875
2ad38827d845, not stripped
```

# Process

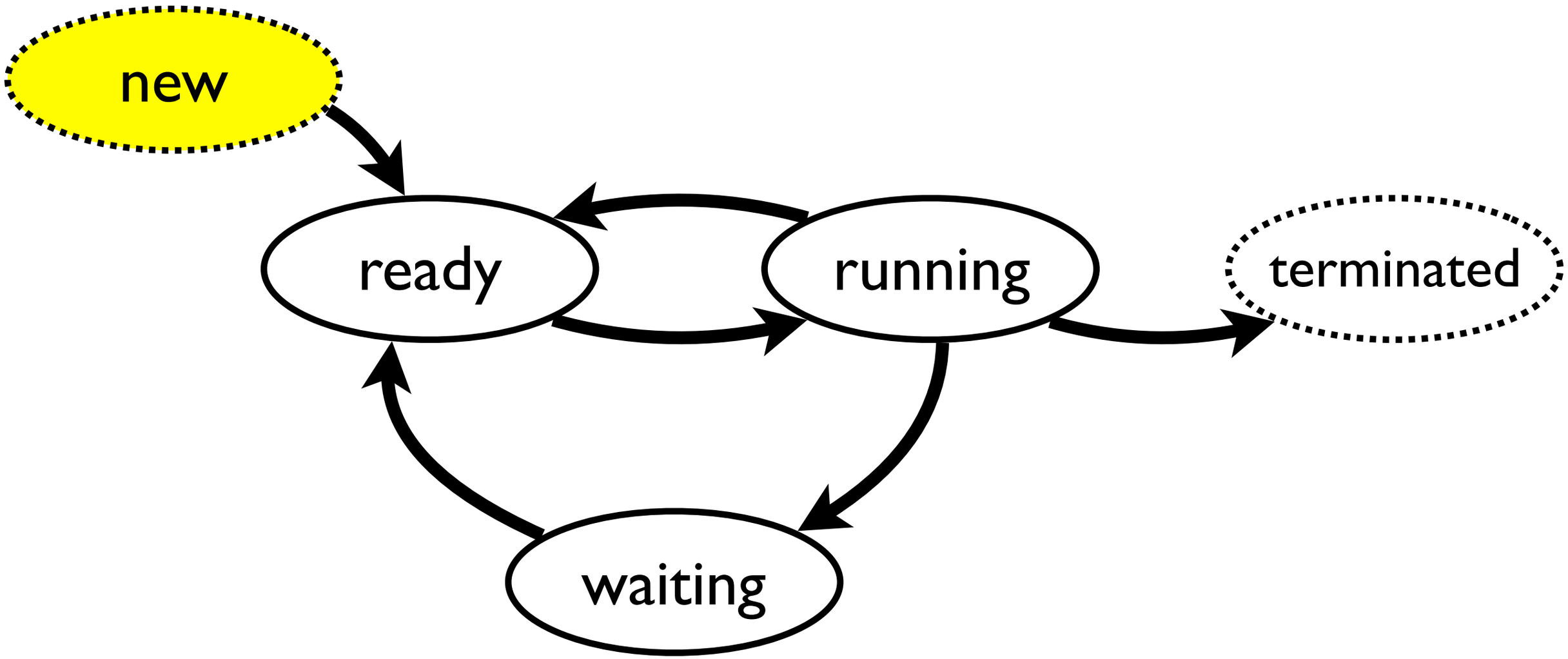
To run the program, the operating system must first create a process.

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

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

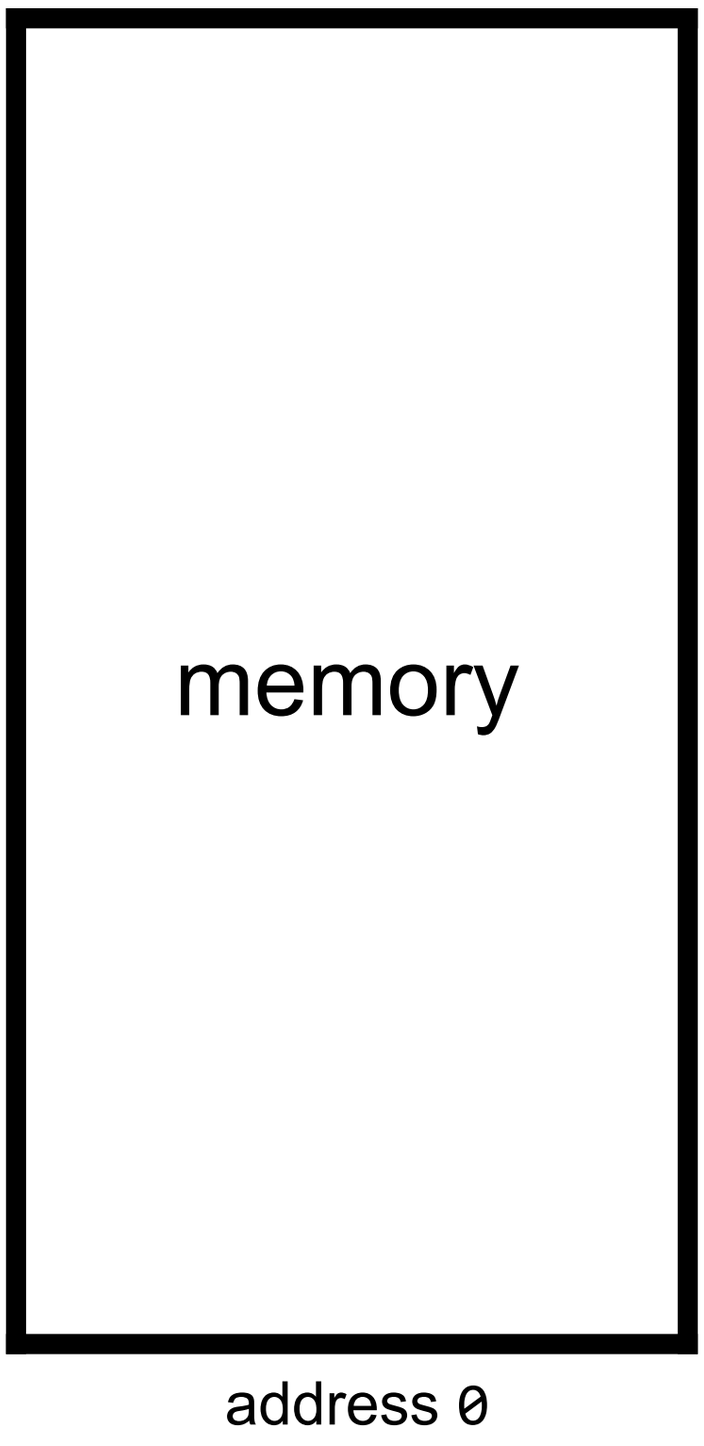
# New

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



# Static memory allocation

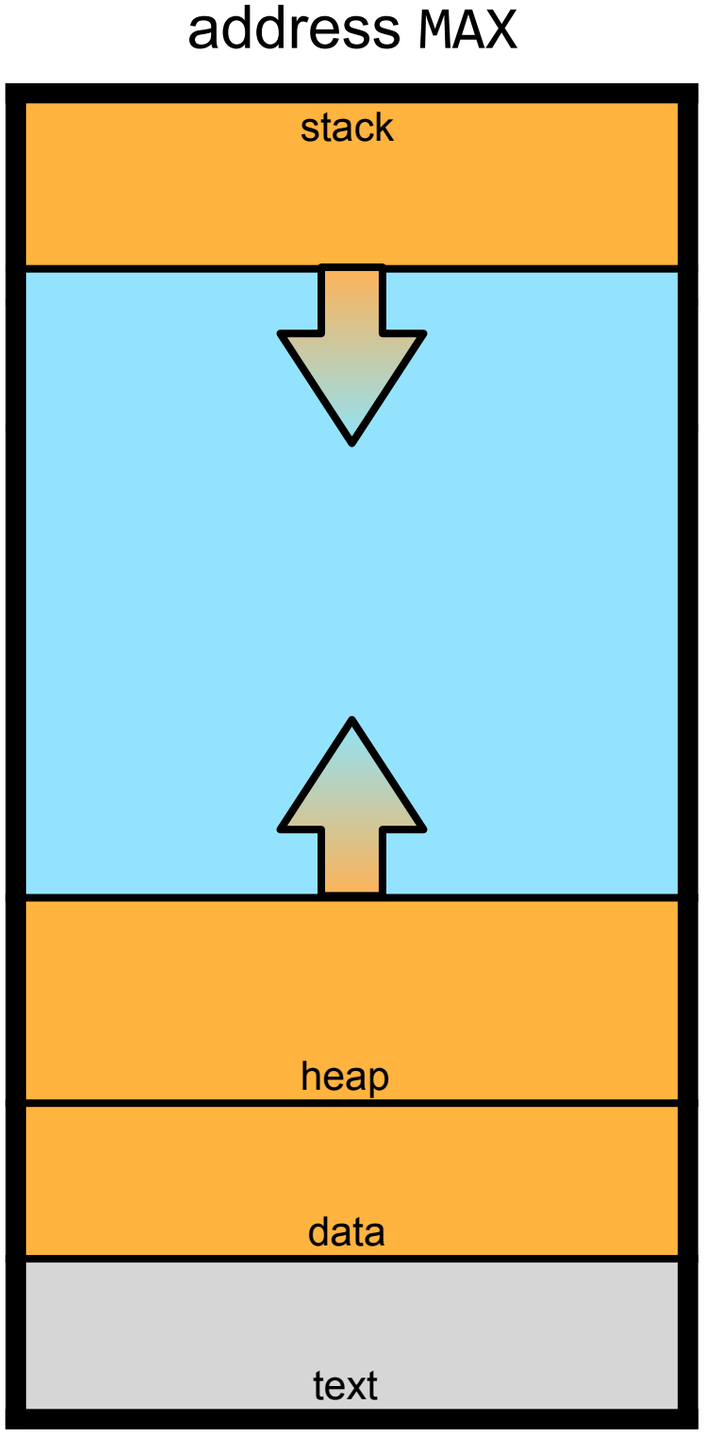
```c
// Example C program

#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);
}
```

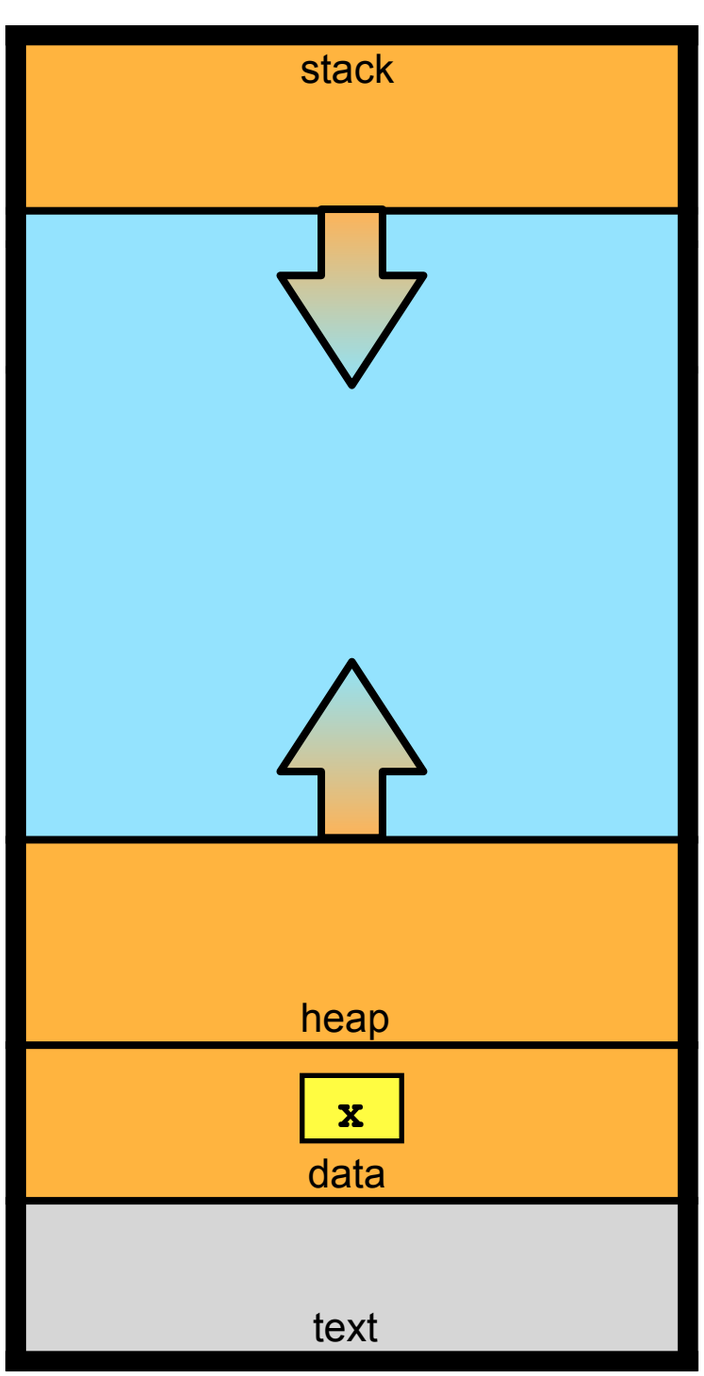



The allocated memory is divided into the following segments:

- text
- data
- heap
- stack

# Static memory allocation

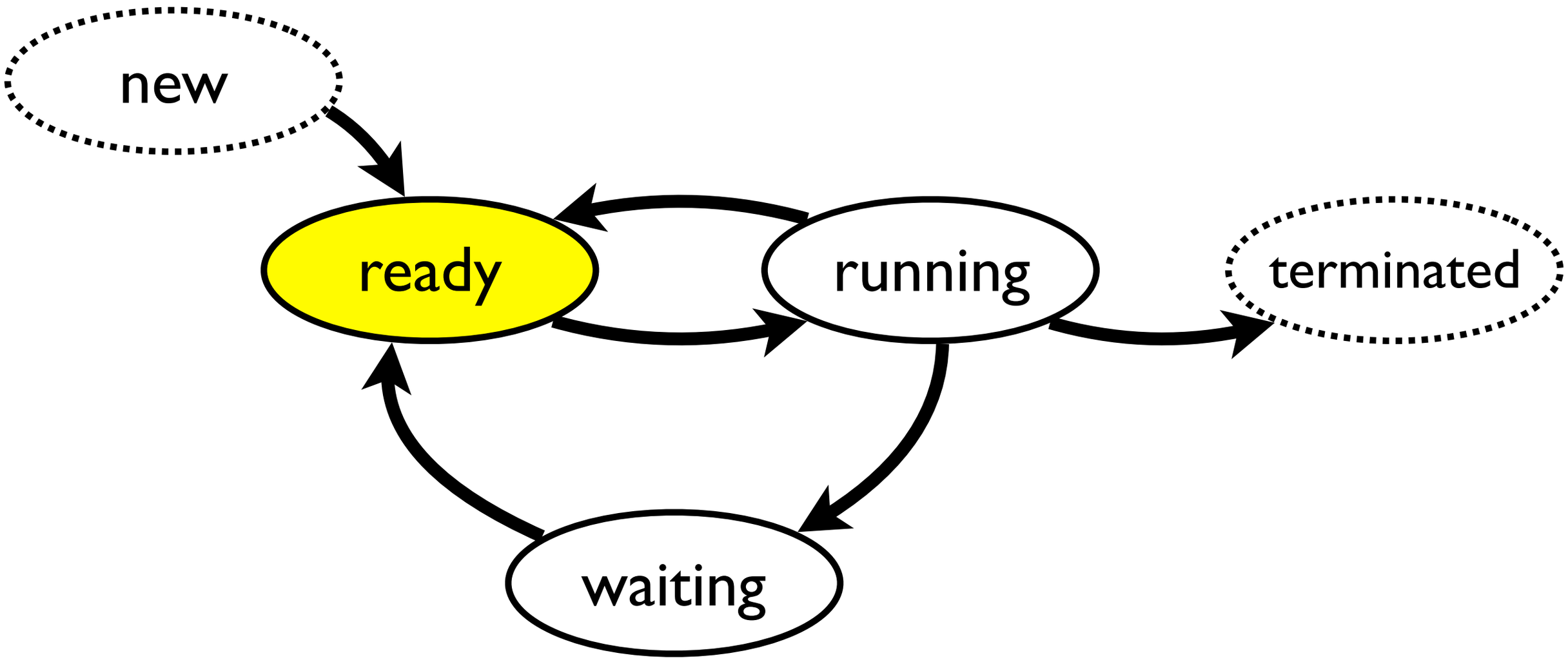
```c
// Example C program

#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**.

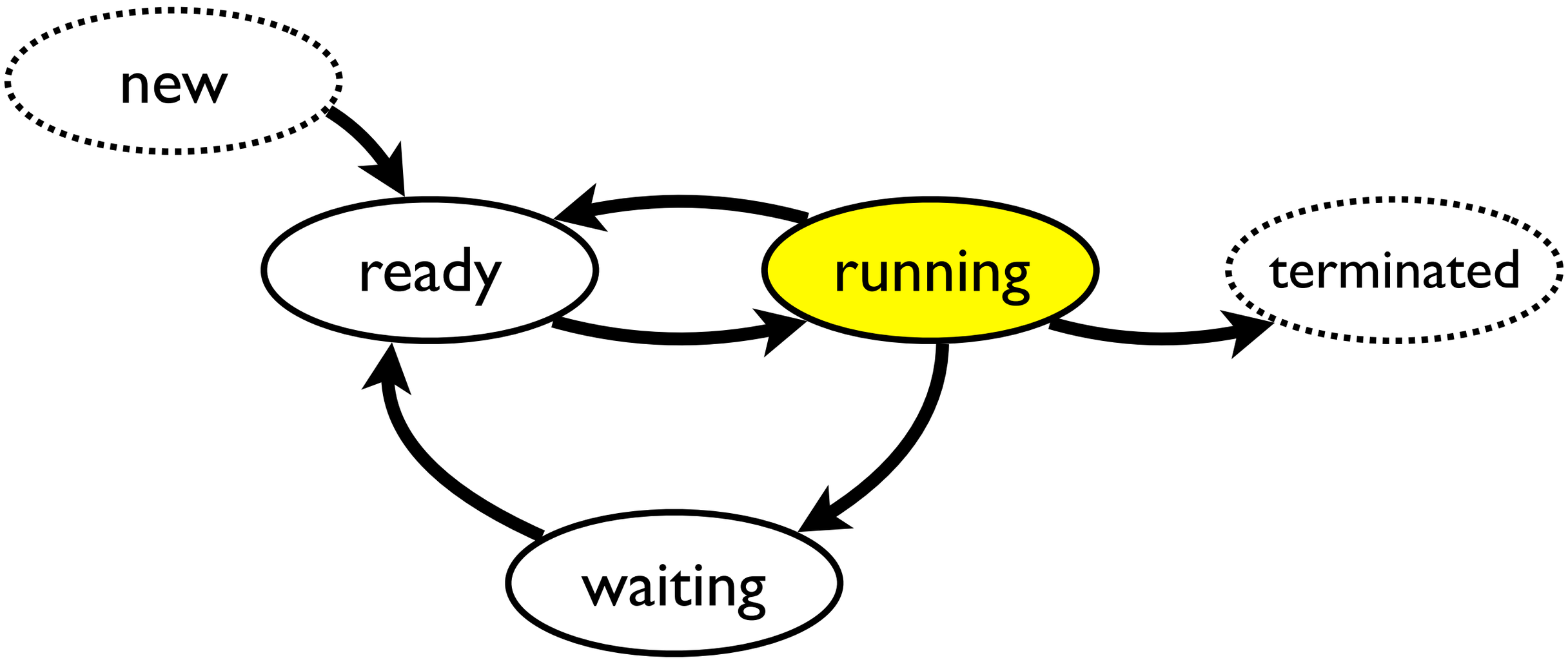
# Ready

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



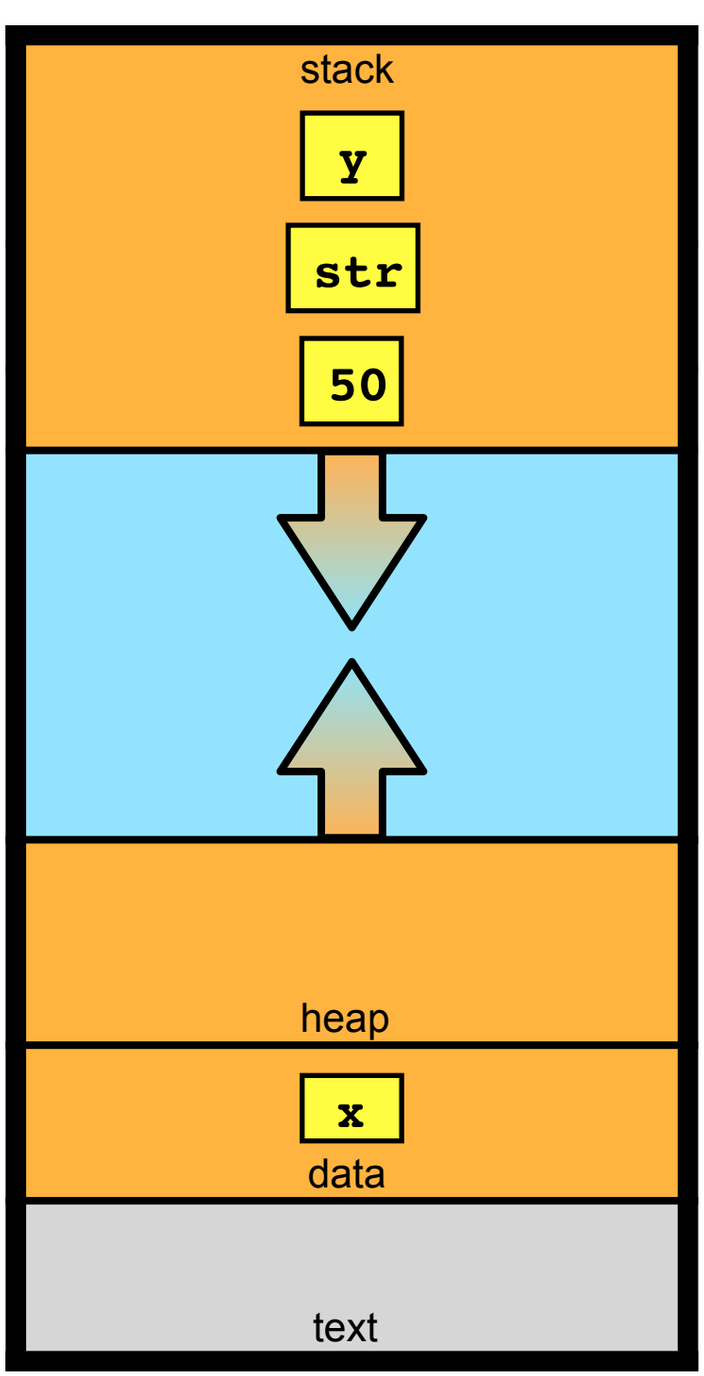
# Running

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);
}
```



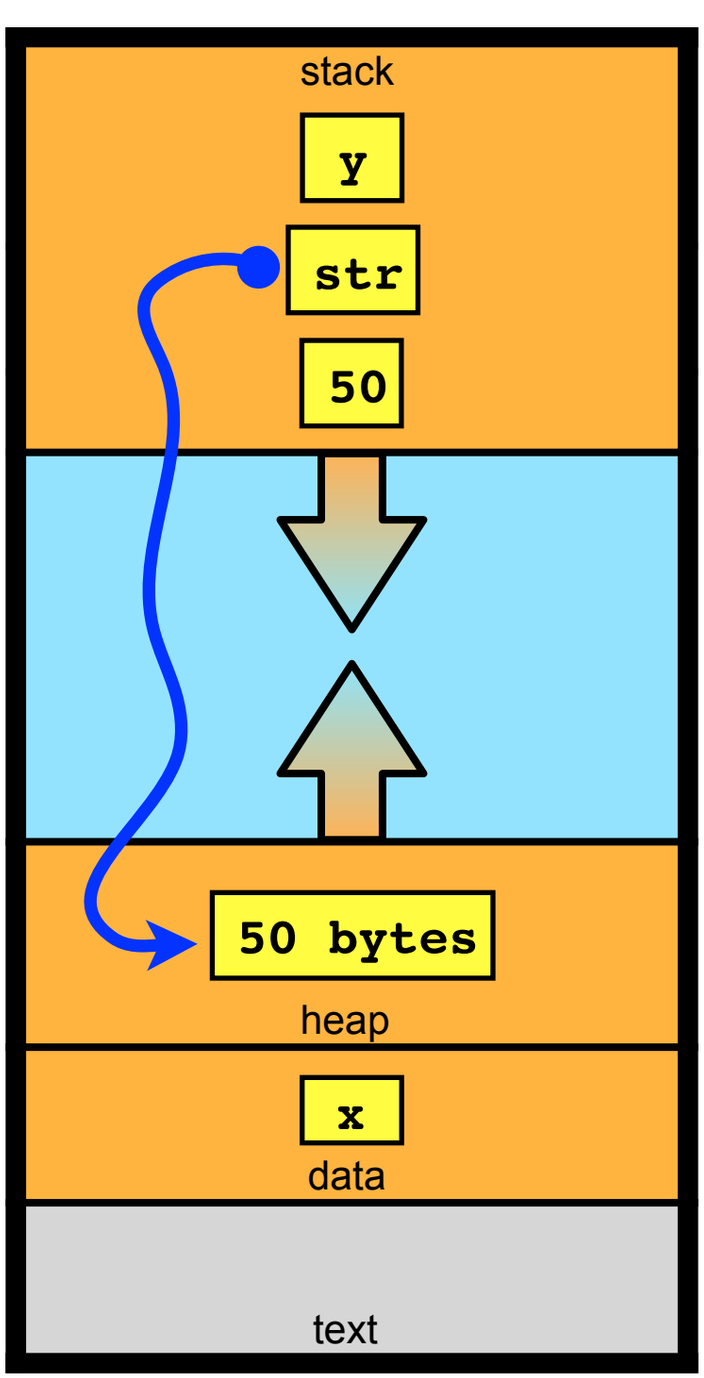

The size of the needed storage for the **local variables** `y` and `str` are also known at compile time but these are only needed within main and storage is allocated on the **stack**.

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);
}
```

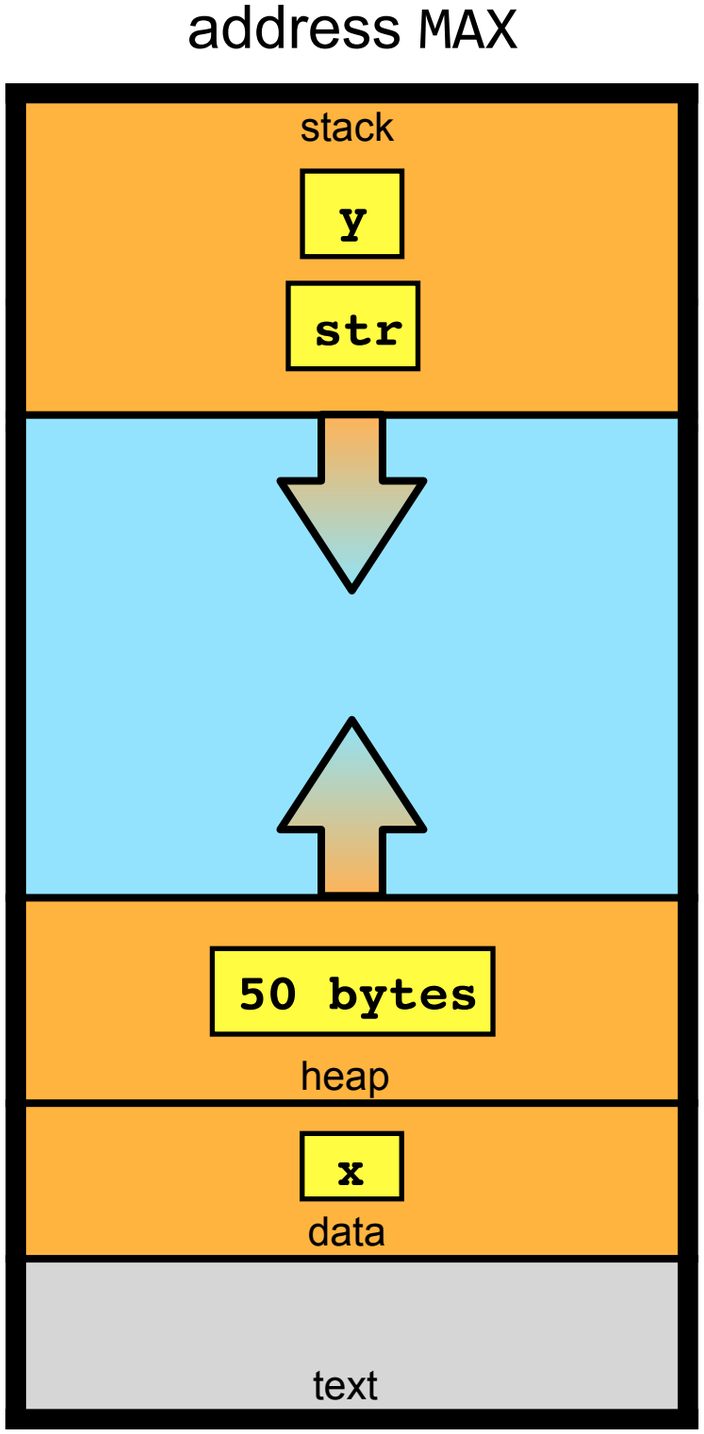



Now **`malloc`** has **dynamically** allocated memory on the heap.

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

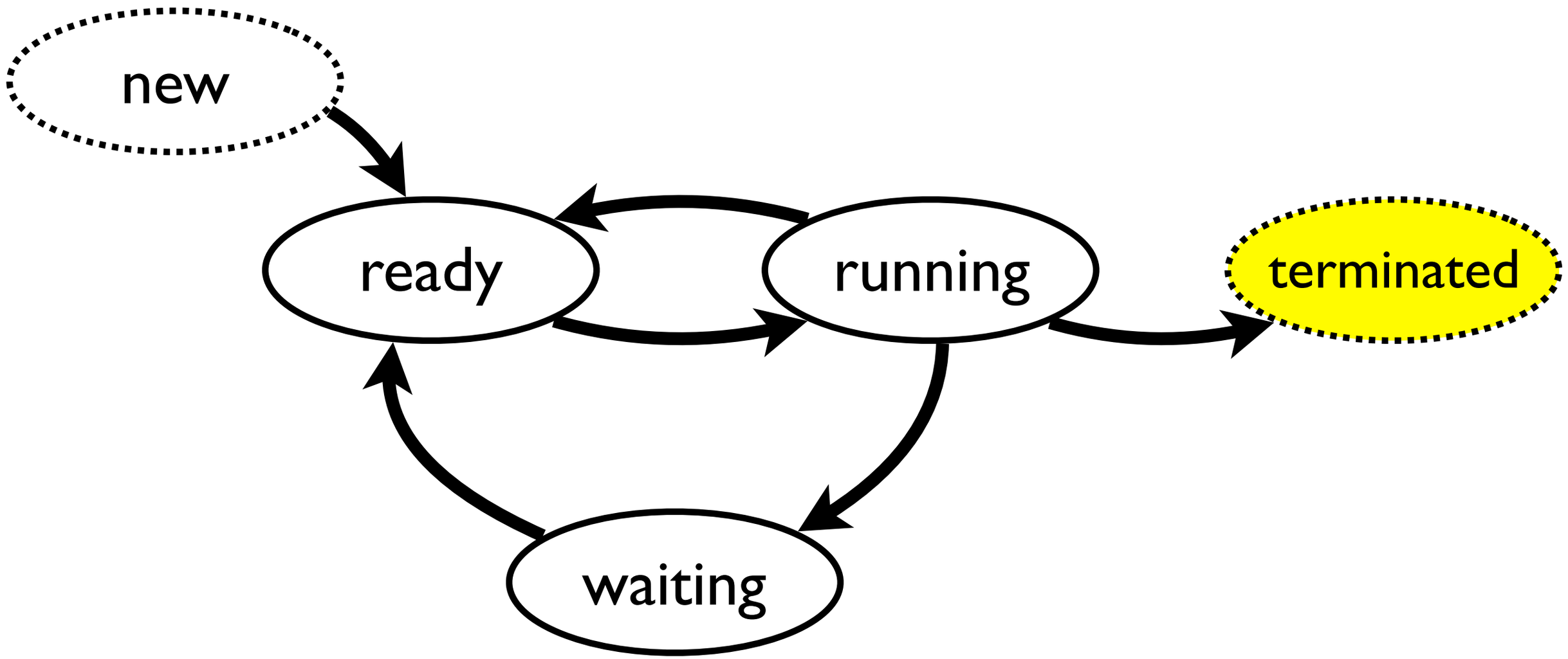
```c
// Example C program

#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.

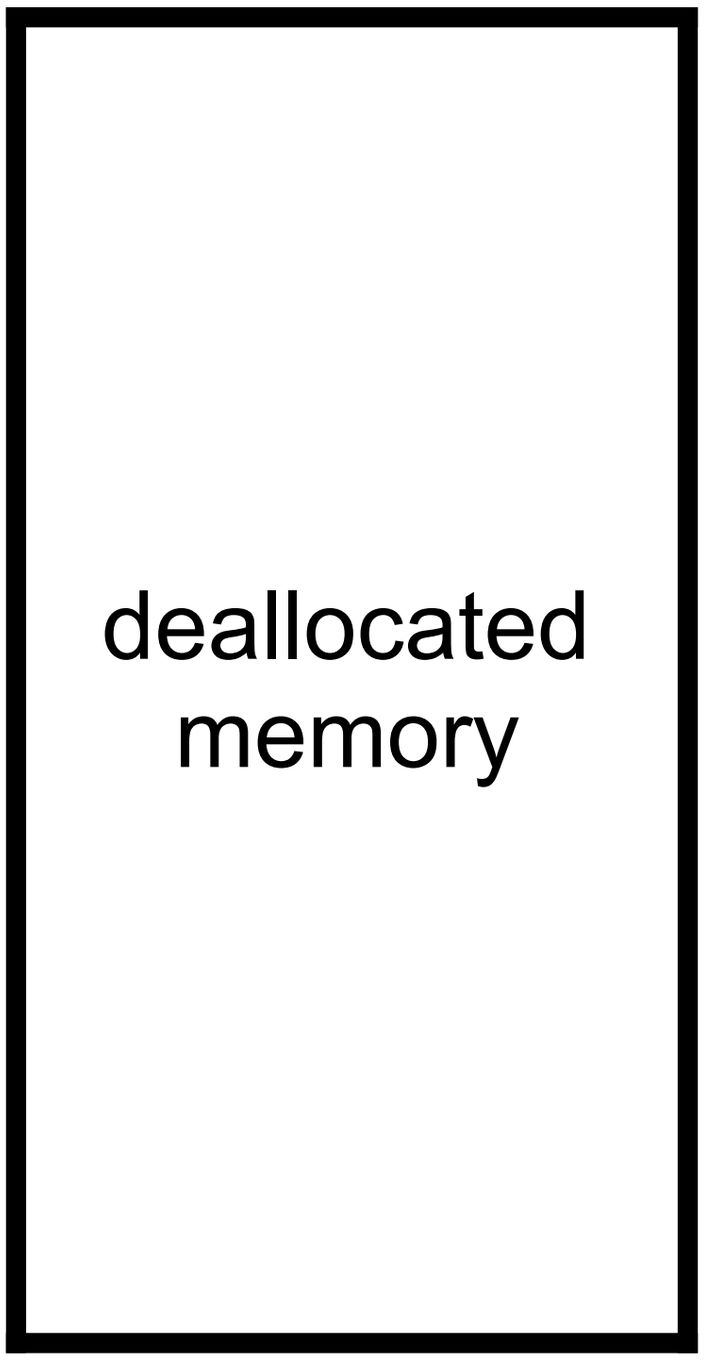
# Termination

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.

Source https://en.wikipedia.org/wiki/Process_control_block 2018-01-21

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 (1)

In computing, a context switch is the process of storing the state of a process or of a thread, so that it can be restored and execution resumed from the same point later. This allows multiple processes to share a single CPU, and is an essential feature of a multitasking operating system.

# Context switch (2)

It is not enough to save and restore the CPU context. Other information, for example the process state must also be saved and restored.

# Context switch (3)

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**.

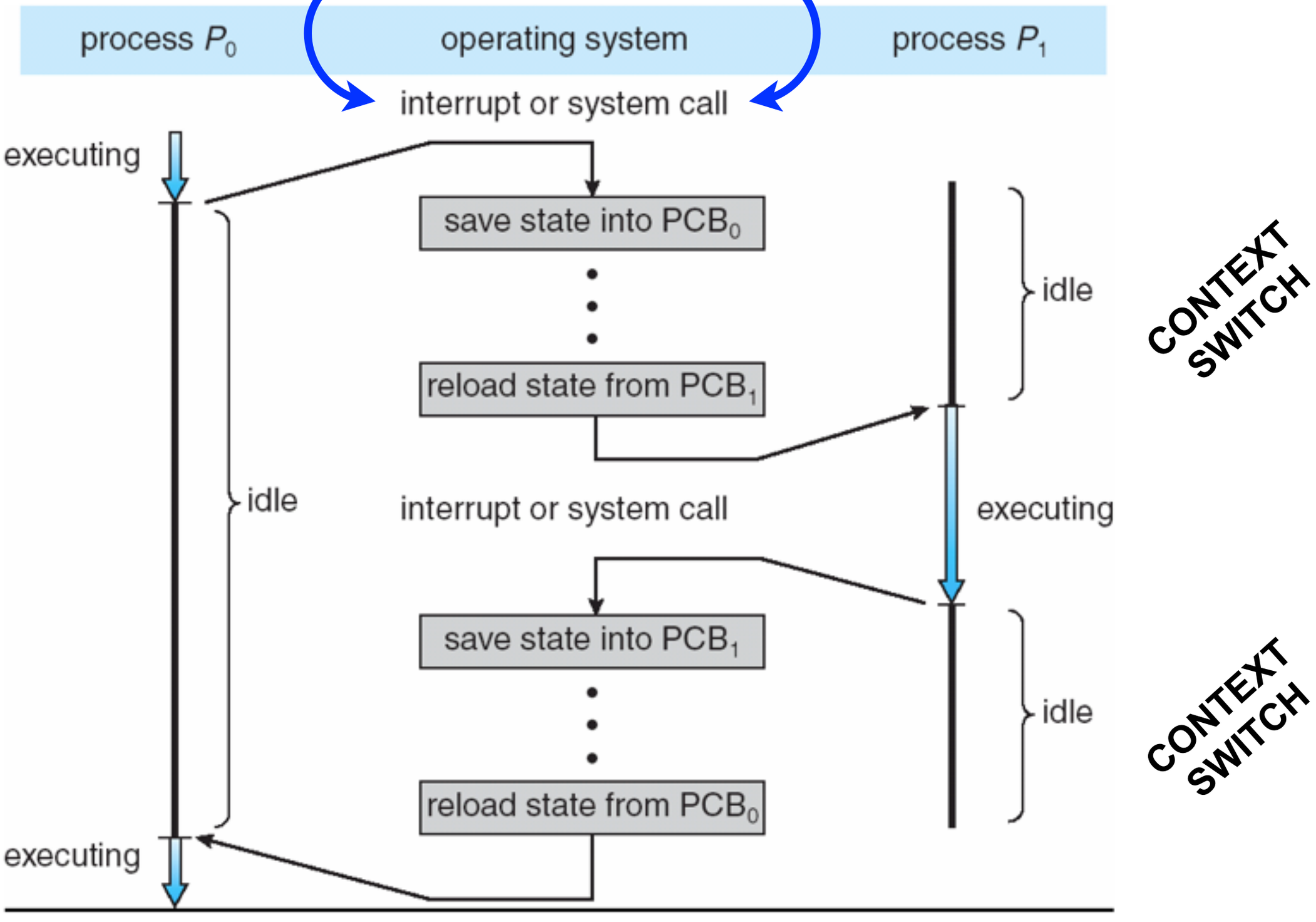
**Context** of a process is represented in the PCB.

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

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

Exception or interrupt.

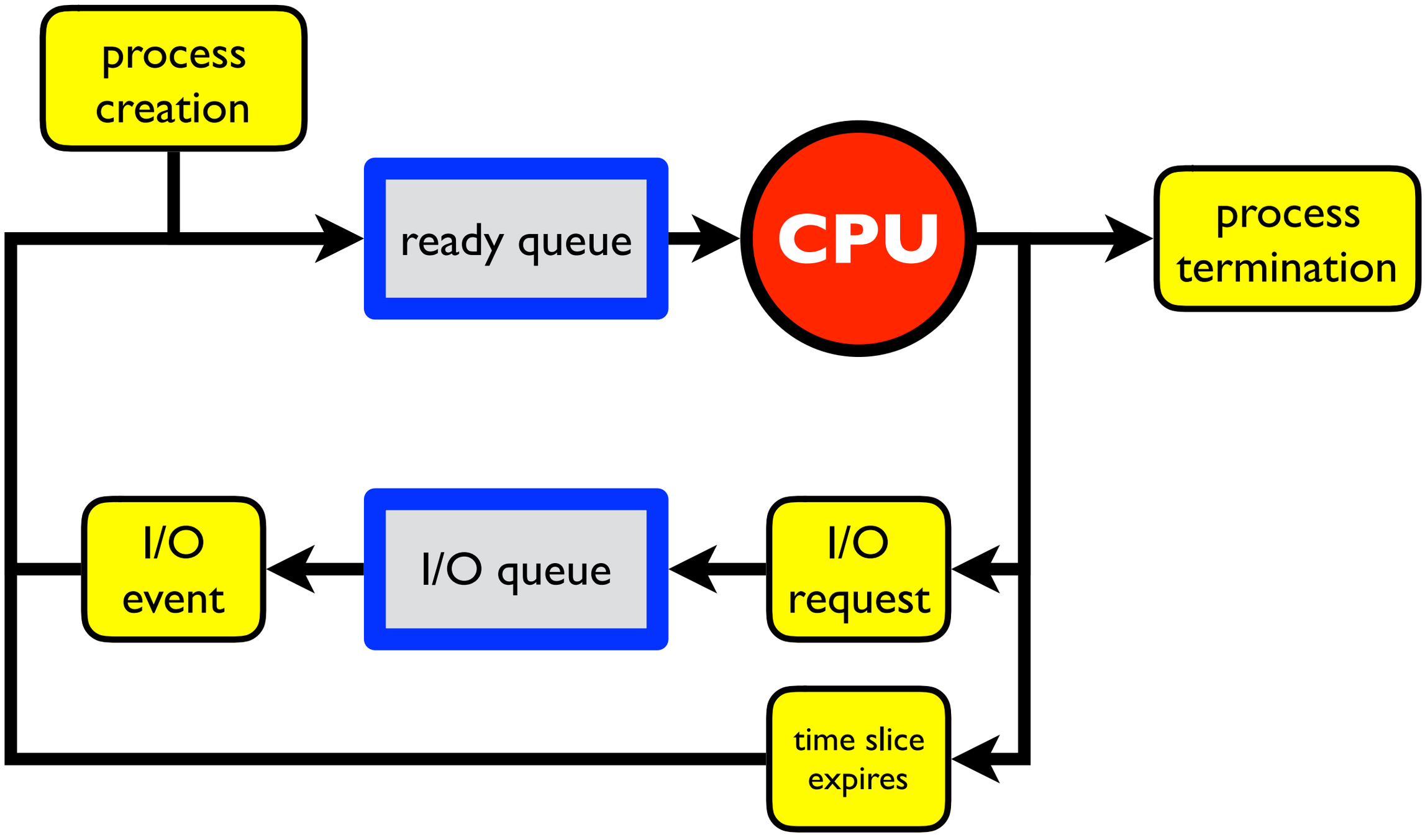
A system call is initiated using a system call exception.



# Process queues

# Process queues

We have already seen two examples of process queues: the ready queue and the I/O queue.



# Process queues

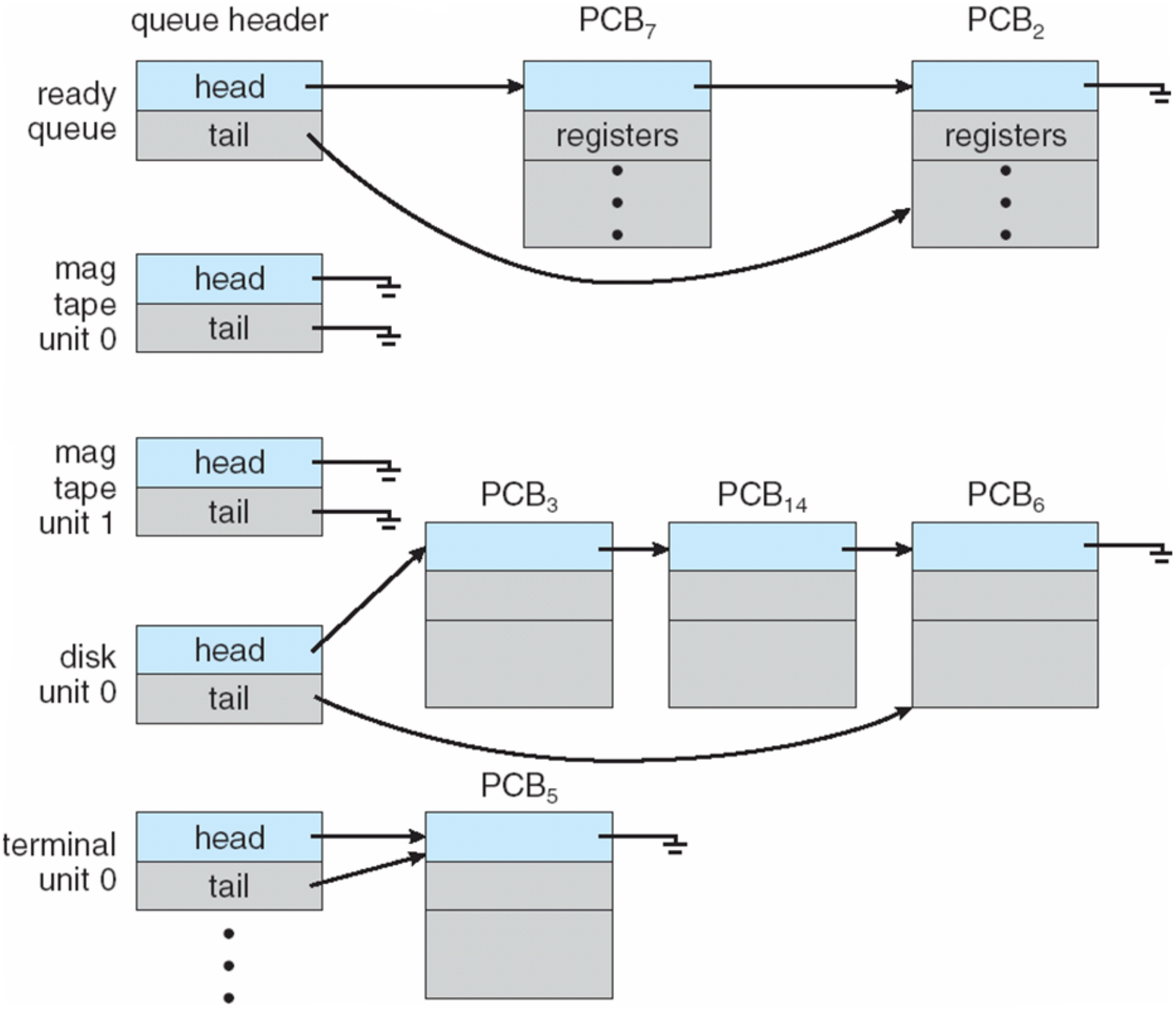
In general we may use one queue for each I/O device.

| | |
|---|---|
| **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 can be formed by linking PCBs together

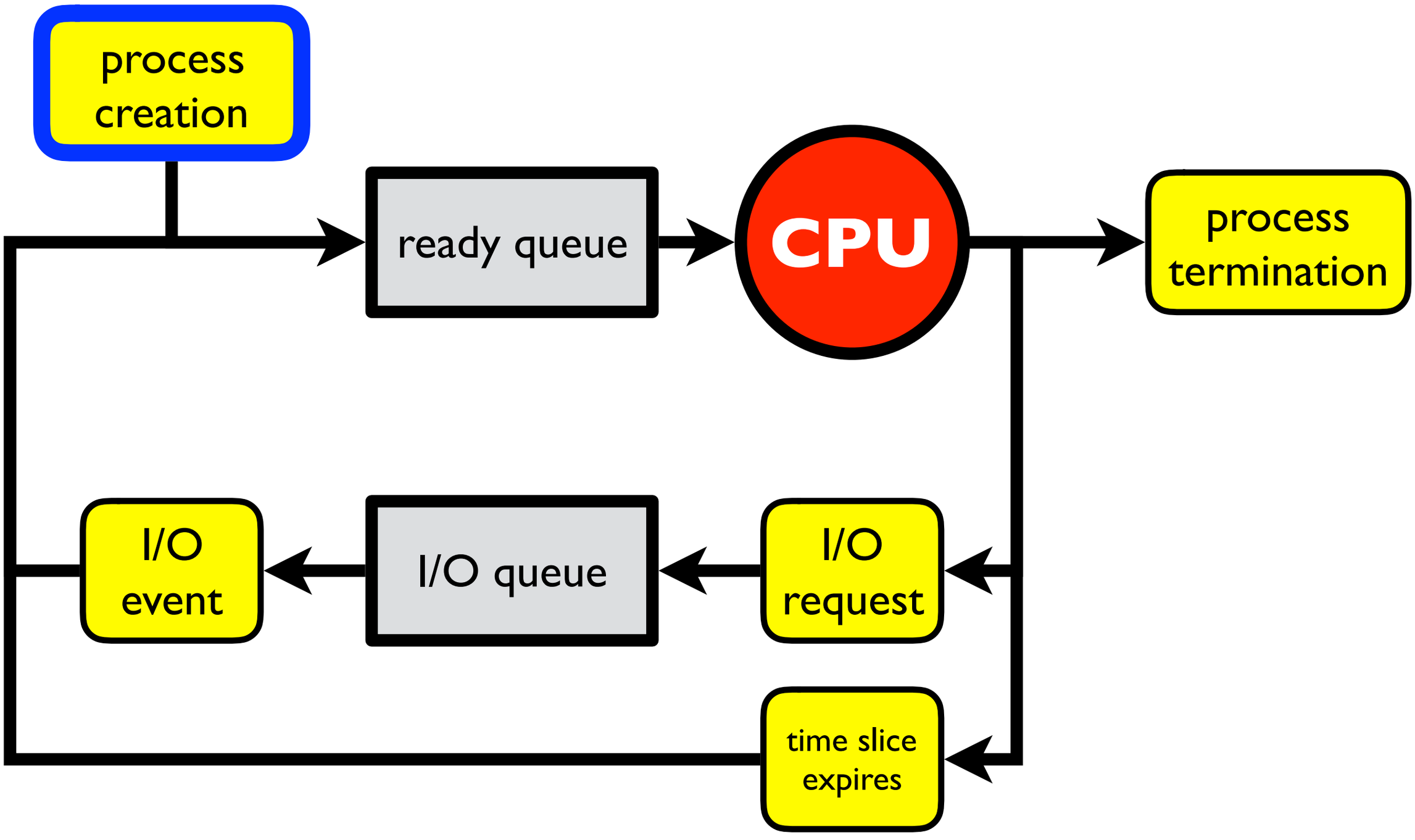
**Ready Queue**

**Device queues**



# Process creation

# How are process created?



# System and Application Programs

## Operating System

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

## Bootstrap program

## Kernel

# Computer Hardware

# 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**.

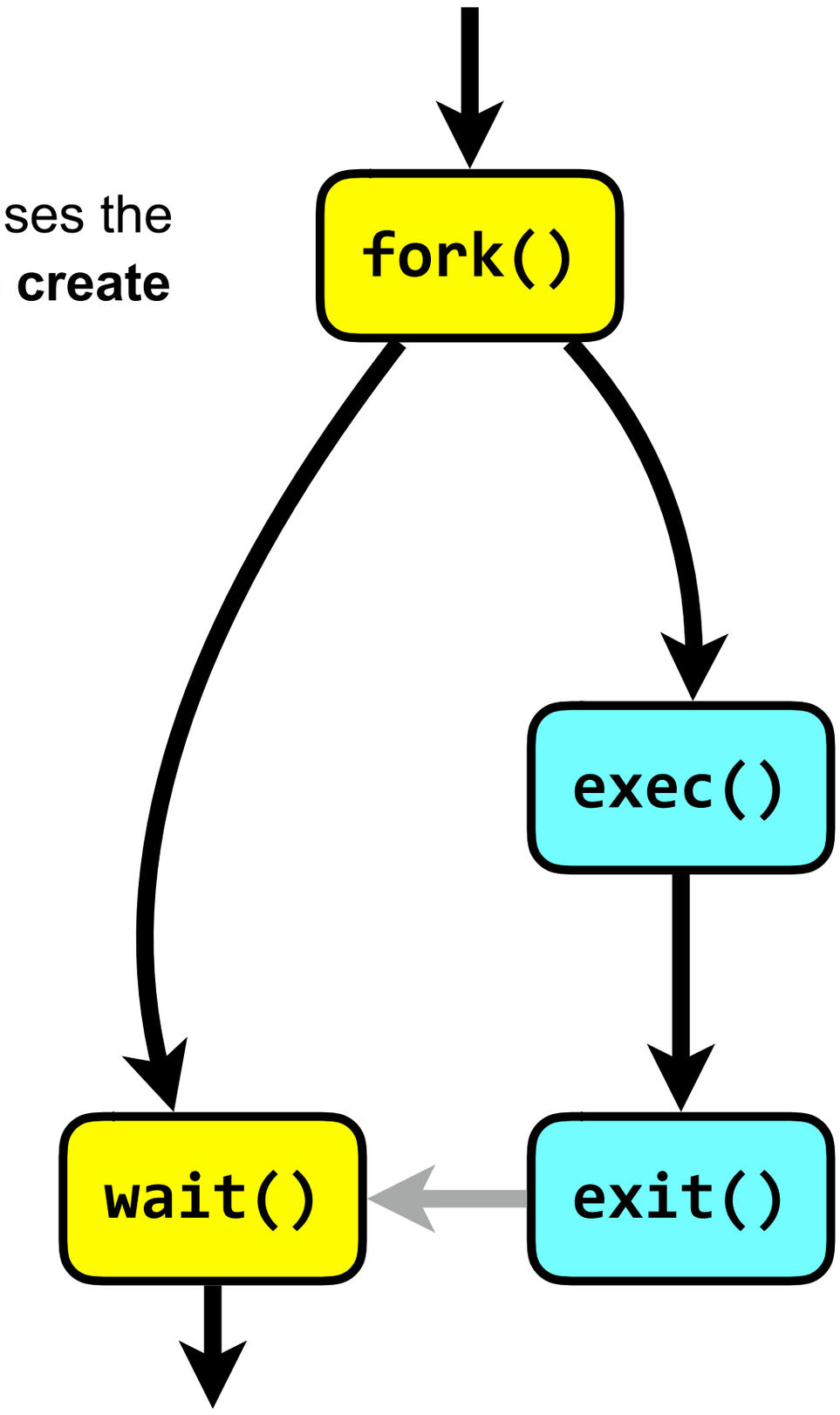
Waits for some **event** to occur ...

# Computer Hardware

# Process creation in Unix/Linux

A single **parent** process executes.

The **parent** process uses the `fork()` system call to **create** a new **child** process.




A new process, the child process is created. The new **child** is a **copy of** its **parent**.

The **child** may use the `exec()` system call after a fork to replace the process's memory space with a new program.

The **child** uses the `exit()` system call to terminate.

The **parent** may use the `wait()` system call to wait for the child to terminate.

The single **parent** continue to execute.

# `fork()`

In Unix-like operating system, fork is an operation whereby a process **creates a copy** of itself.

- ★ The process calling fork() is called the **parent**.
- ★ The new process is called the **child** of the parent.
- ★ Fork is usually a **system call**, implemented in the kernel.
- ★ Fork is the primary (and historically, only) method of process creation on Unix-like operating systems.

# exec()

In Unix-like operating systems, the **exec family of system calls** runs an executable file in the context of an already existing process, **replacing** the previous **executable**.

- ★ As a new process is not created, the process identifier (PID) does not change, but the machine code, data, heap, and stack of the process are replaced by those of the new program.
- ★ A file descriptor open when an exec call is made remain open in the new process image, unless was fcntled with FD_CLOEXEC. This aspect is used to specify the standard streams (stdin, stdout and stderr) of the new program.
- ★ As a result of exec, all data in the old program that were not passed to the new program, or otherwise saved, become lost.

# `wait()`

A **parent** process can use the wait() system call to suspend its execution until one of its children terminates and get the exit status of the terminated child.

```
pid_t wait(int *status);
```

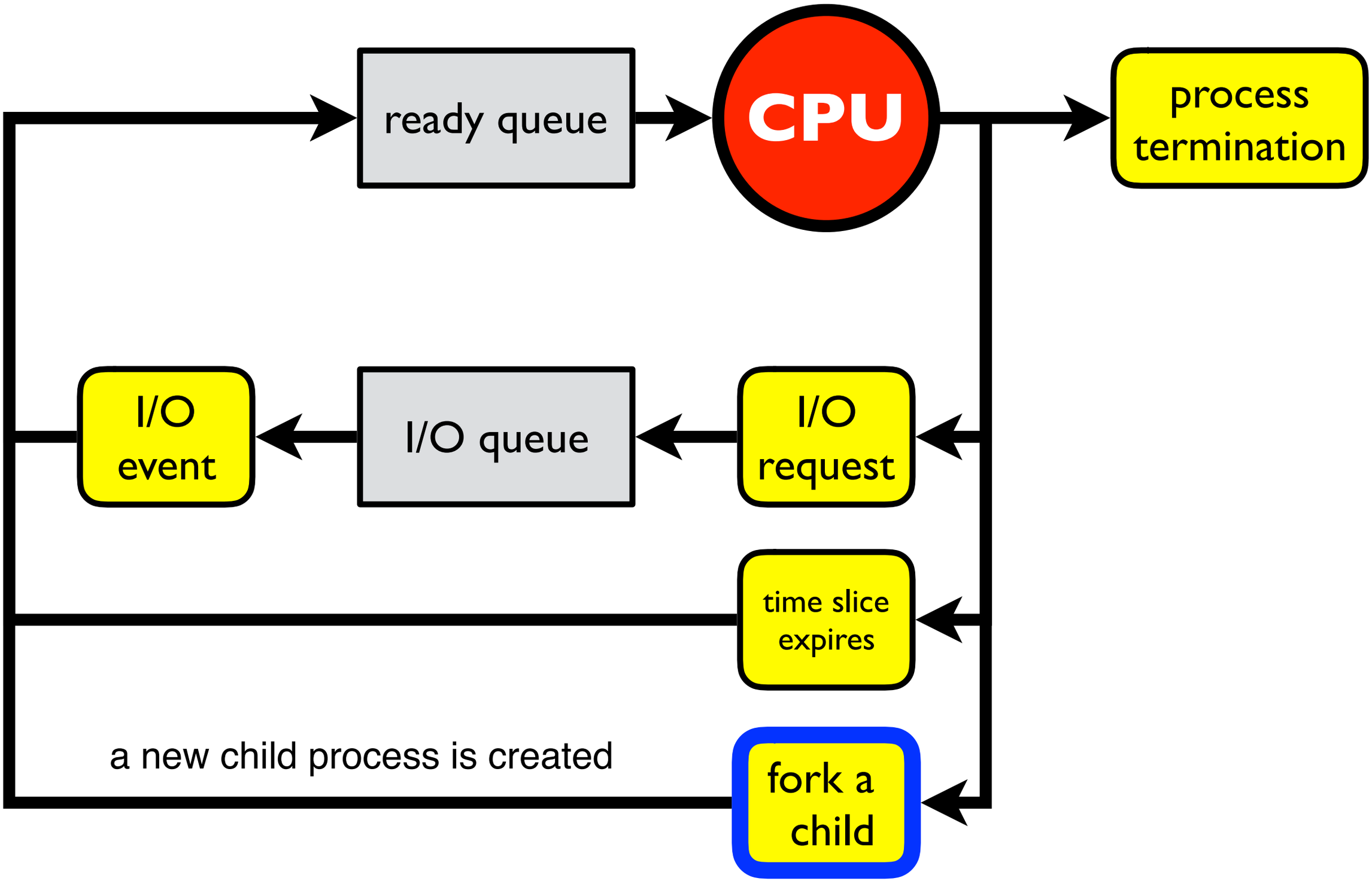
- ★ If there is **no child** process running when the call to wait() is made, then this wait() has **no effect** at all.
- ★ Otherwise, executing wait() **suspends the caller until one of its children terminates**.
- ★ **Returns** the **PID** of the terminated **child** process.
- ★ If status is not a null pointer, the **exit status of the terminated child** will be stored at the location pointed to by status.

# `exit()`

When a process terminates it executes an exit() system call, either directly in its own code, or indirectly via library code.

```
void exit(int status);
```

- ★ The exit() call has **no return value** as the process that calls it terminates and so couldn't receive a value anyway.
- ★ The exit() call **resumes** the execution of a **waiting parent** process.
- ★ The exit() call also **communicates** the **status** parameter value **to** the **parent** process who can get the status using the wait() system call.


ready queue
CPU
process termination
I/O event
I/O queue
I/O request
time slice expires
a new child process is created
fork a child

# Create a new process using fork()

User Space

Kernel Space

**Parent Process**

TEXT (instructions)
DATA

Parent calls fork()

Resources (open files, etc)

**Child Process**

TEXT (instructions)
DATA

The child process is a copy of the parent

Resources (open files, etc)

OS creates a new process - a child process

The child process is a copy of the parent (copy of TEXT, DATA and Resources)

User Space

Kernel

**Parent Process**

**TEXT (instructions)**
**DATA**

Parent calls fork()

**Resources (open files, etc)**

**Child Process**

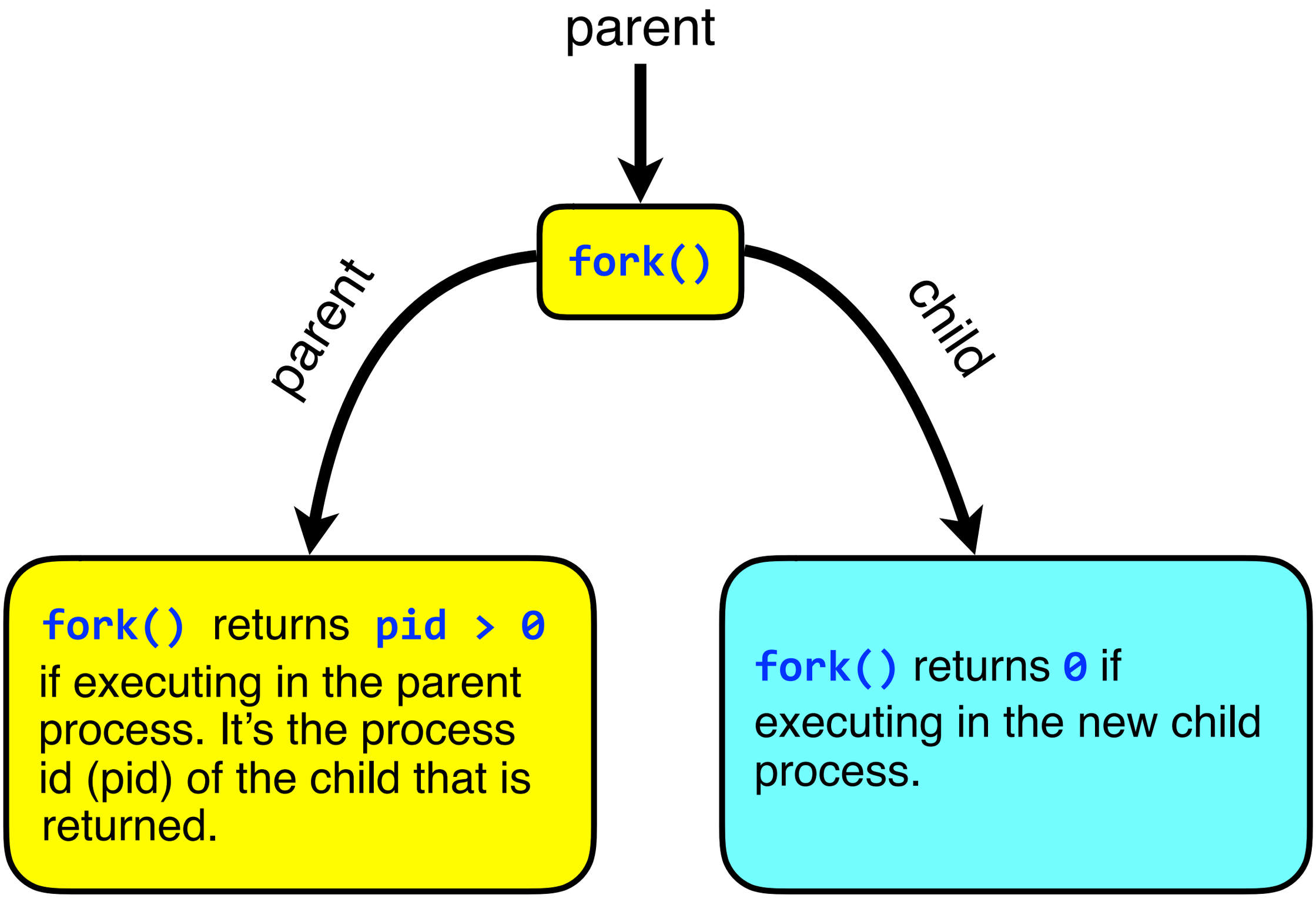
**TEXT (instructions)**
**DATA**

The child process is a copy of the parent

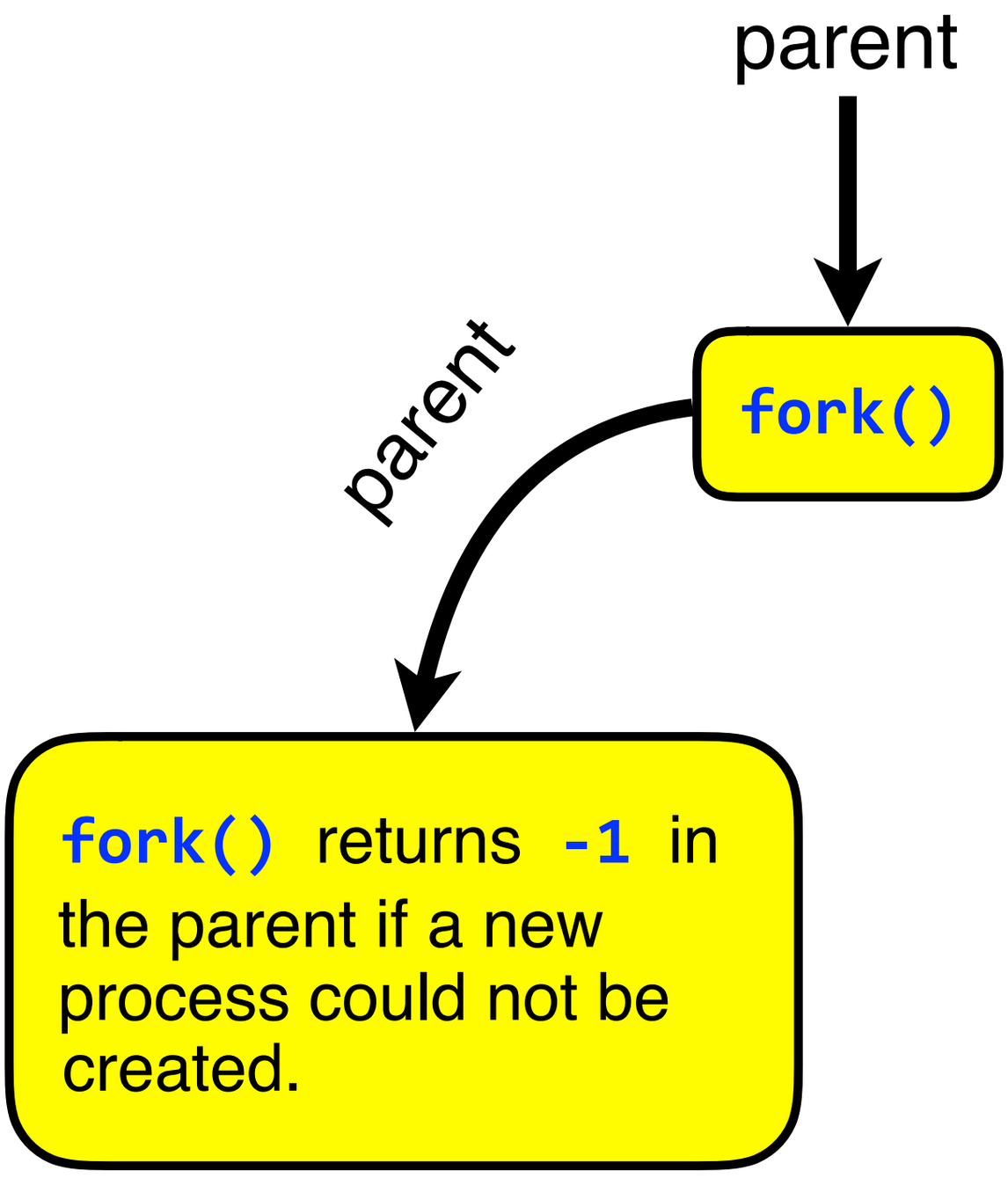
**Resources (open files, etc)**

After **`fork()`**, parent and child will execute the same program (TEXT). How can they be distinguished from each other?

# On success, `fork()` return twice!



# On failure, `fork()` returns `-1` in the parent



# Parent Process

STACK

```
pid = 766611
```

TEXT

```
pid = fork();

switch (pid) {
case -1:
  // ERROR
  exit(EXIT_FAILURE);
case 0:
  // CHILD
  exit(EXIT_SUCCESS);
default:
  // PARENT
  exit(EXIT_SUCCESS);
}
```

# Child Process

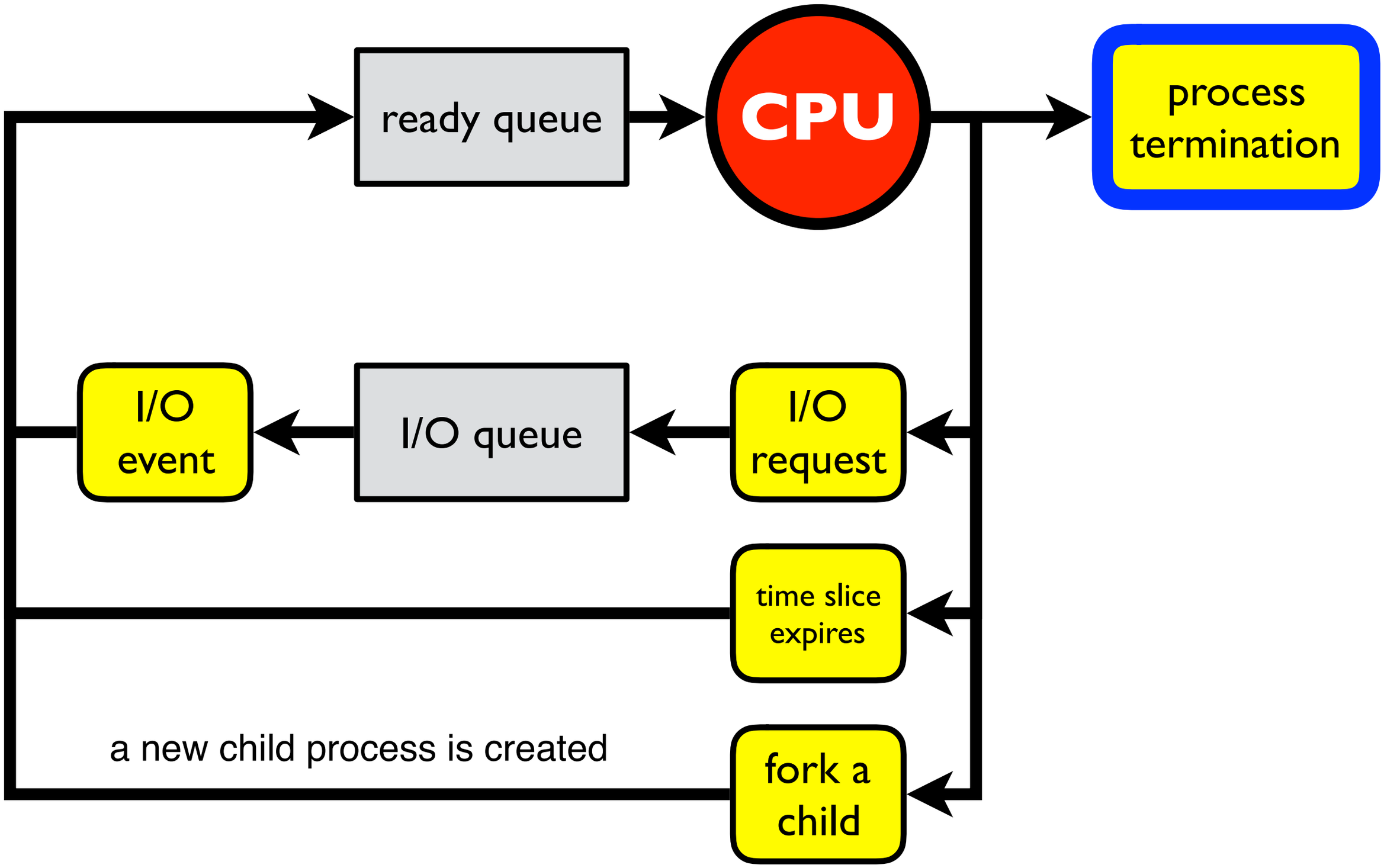
STACK

```
pid = 0
```

TEXT

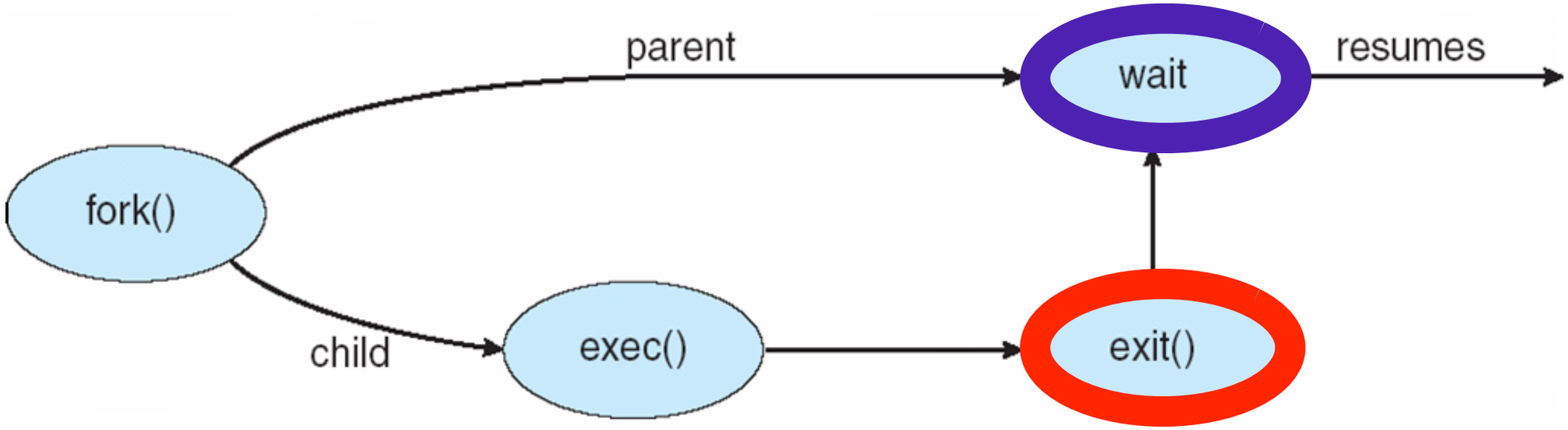
```
pid = fork();

switch (pid) {
case -1:
  // ERROR
  exit(EXIT_FAILURE);
case 0:
  // CHILD
  exit(EXIT_SUCCESS);
default:
  // PARENT
  exit(EXIT_SUCCESS);
}
```

# Process termination


ready queue
CPU
process termination
I/O event
I/O queue
I/O request
time slice expires
a new child process is created
fork a child

# `exit()` and `wait()`

When a process terminates it executes an `exit()` system call, either directly, or indirectly via library code. This leaves an **exit status** value (typically an integer) **in** the **PCB** of the terminated process for the parent process to read later.




The parent may use the `wait()` system call to wait for the child to terminate and read the exit status value of the terminated child process.

# Zombie processes

# Zombie

A terminated process is said to be a zombie or defunct until the parent does `wait()` on the child.

- ★ When a process terminates all of the memory and resources associated with it are deallocated so they can be used by other processes.
- ★ However, the **exit status is maintained in the PCB until the parent picks up the status using** `wait()` and deletes the PCB.
- ★ A child process always first becomes a zombie before being removed from the resource table.
- ★ In most cases, under normal system operation zombies are immediately waited on by their parent.
- ★ Processes that stay zombies for a long time are generally an error and cause a resource leak.

**Source:** https://en.wikipedia.org/wiki/Zombie_process

# Orphan processes

# Orphan

An orphan process is a process whose parent process has terminated, though it remains running itself.

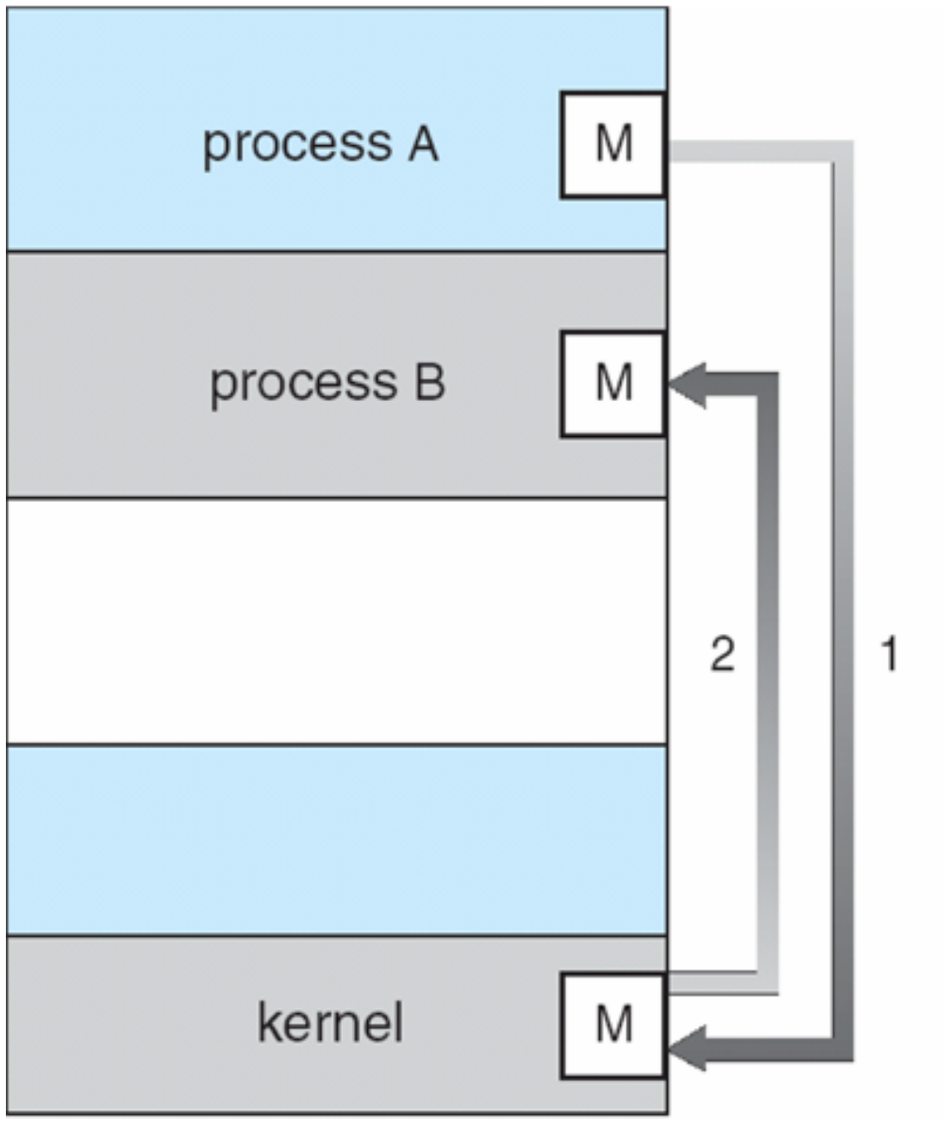
- ★ Any orphaned process will be immediately **adopted by** the special **init** system process.
- ★ Even though technically the process has the **init** process as its parent, it is still called an orphan process since the process that originally created it no longer exists.
- ★ A zombie process is not the same as an orphan process. An orphan process is a process that is still executing, but whose parent has died. An orphan process do not become zombie processes; instead, they are adopted by init (process ID 1), which automatically waits on all its child process.

# Inter process communication (IPC)

# How can we make processes communicate (sharing information)?

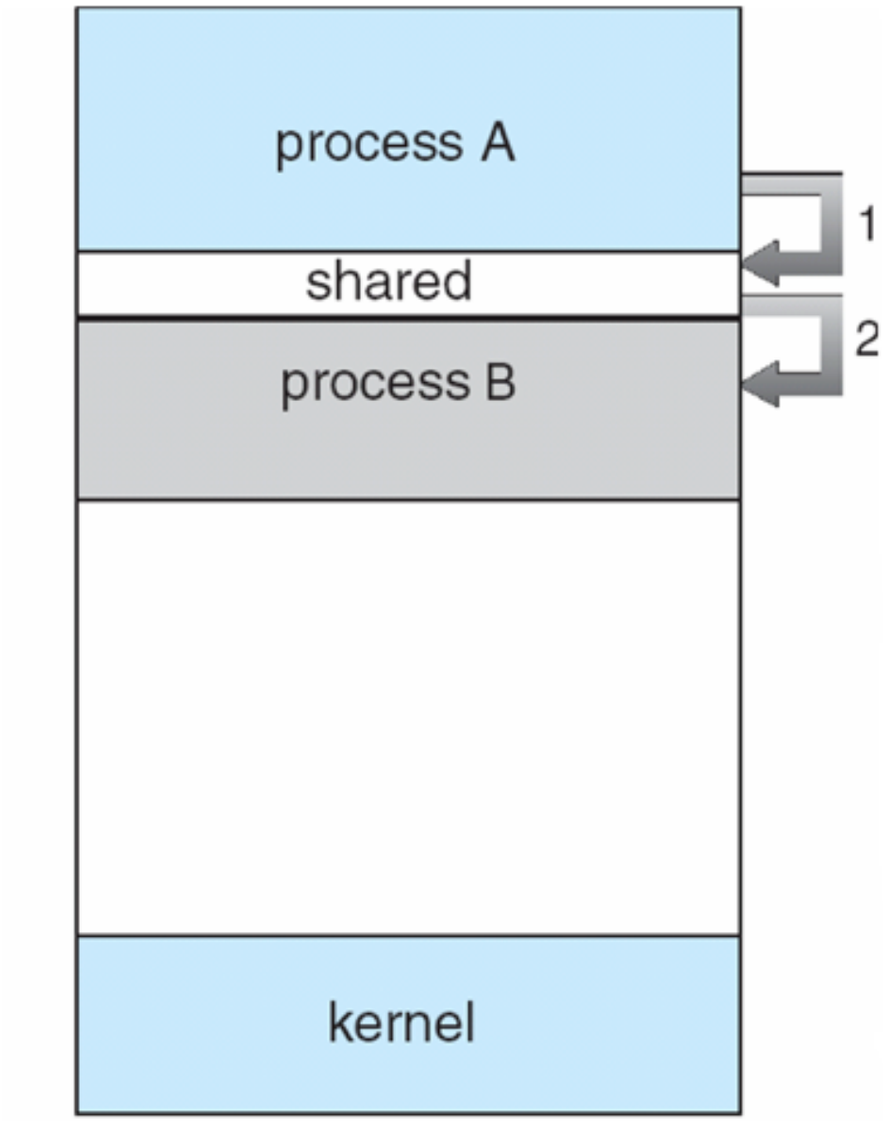
Two methods: **message passing** and **shared memory**.

# 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.

# Signals

# Signals (1)

Signals are a limited form of inter-process communication used in Unix, Unix-like, and other POSIX-compliant operating systems

- ★ A signal is an **notification** sent to a process in order to notify it of an event that occurred.
- ★ When a signal is sent, the operating system **interrupts** the target process's **normal flow of execution** to deliver the signal.
- ★ If the process has previously **registered** a **signal handler**, that routine is executed. Otherwise, the **default signal handler** is executed.

# Signals (2)

Signals can be synchronous or asynchronous.

**Synchronous signals**

- ★ Are delivered to the same process that performed the operation that caused the signal.
- ★ Examples of synchronous signals: **illegal memory** access and **division by zero**.

**Asynchronous signals**

- ★ Are generated by an event external to a running process.
- ★ Typically, an asynchronous signal is sent to another process.
- ★ Examples of asynchronous signals: **terminating** a process (ctrl-c), **timer expire**.

# Signals (3)

Typing certain key combinations at the controlling terminal of a running process causes the system to send it certain signals

- **Ctrl-C** sends an INT signal (**SIGINT**); by default, this causes the process to **terminate**.

- **Ctrl-Z** sends a TSTP signal (**SIGTSTP**); by default, this causes the process to **suspend** execution.

# Signals (4)

The kernel can generate a signal to notify the process of an event.

- ★ **Exceptions** such as **division by zero** or a **segmentation violation** will generate signals.

- ★ Divison by zero will generate a **SIGFPE** signal.

- ★ Segmentation violation will generate a **SIGSEGV** signal.

- ★ If not explicitly caught, SIGFPE and SIGSEGV will cause a core dump and a program exit.

# kill()

In Unix-like operating systems, the kill system call is used to send signals to processes.

There are many different signals that can be sent, although the signals in which users are generally most interested are **SIGTERM** and **SIGKILL**.

★ The **SIGKILL** signal is sent to a process to cause it to **terminate immediately** (kill). In contrast to SIGTERM and SIGINT, this signal cannot be caught or ignored, and the receiving process cannot perform any clean-up upon receiving this signal.

★ The **SIGTERM** signal is sent to a process to request its termination. Unlike the SIGKILL signal, it can be caught and interpreted or ignored by the process. This allows the process to perform nice termination releasing resources and saving state if appropriate. SIGINT is nearly identical to SIGTERM.

**Source**: https://en.wikipedia.org/wiki/Kill_(command) 2016-01-25

https://en.wikipedia.org/wiki/Unix_signal 2016-01-25

# 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)

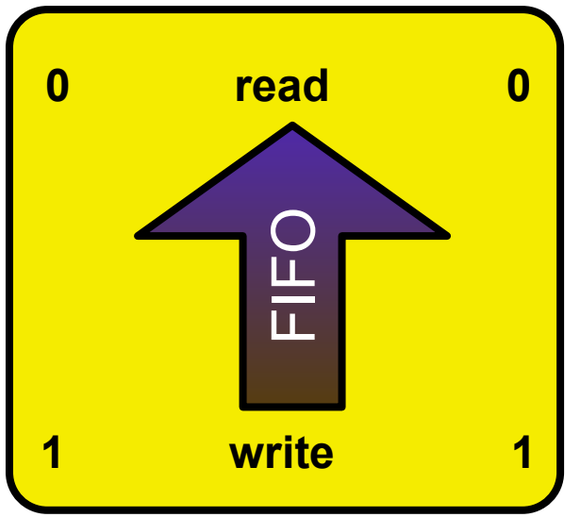
★ As a result, ordinary pipes are **unidirectional**, allowing only **one-way communication**.

★ 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
```

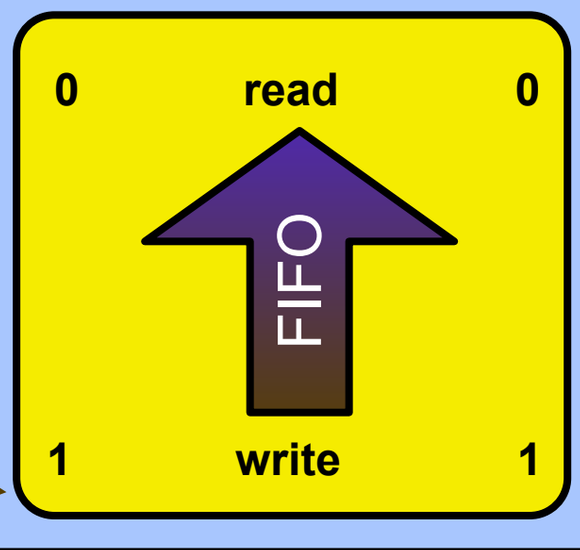
| Descriptors | |
|---|---|
| 0 | stdin |
| 1 | stdout |
| 2 | stderr |
| 3 | pipe read |
| 4 | pipe write |

**(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 Fork

+

# 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.

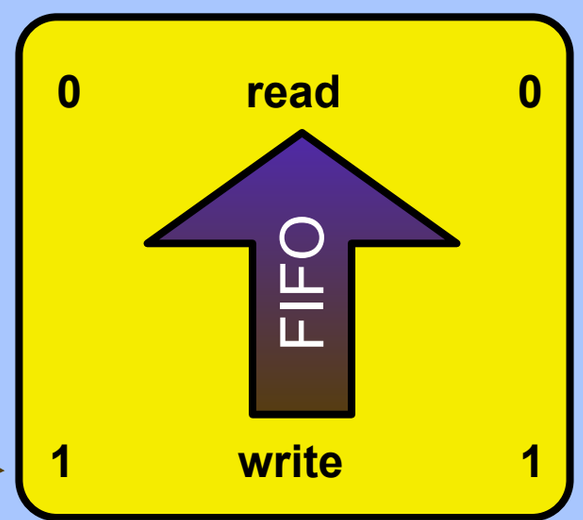
Now, the parent can act as a single producer and the child as a single consumer of data through the pipe using the **`read()`** and **`write()`** system calls - just as if it was a file.

## 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.