Processes

What is a *process*?

A *process* is a program in execution. The components of a process are: the program to be executed, the data on which the program will execute, the resources required by the program—such as memory and file(s)—and the status of the execution.

Is a process the same as a program? No!, it is both *more* and *less*.

- *more*—a program is just part of a process context.
  tar can be executed by two different people—same program (shared code) as part of different processes.
- *less*—a program may invoke several processes.
  cc invokes **cpp**, **cc1**, **cc2**, **as**, and **ld**.

Programming: *uni-* versus *multi-*

Some systems allow execution of only one process at a time (e.g., early personal computers).

They are called *uniprogramming* systems.

Others allow more than one process, i.e., concurrent execution of many processes. They are called *multi-programming* (NOT *multiprocessing*!) systems.

In a multiprogramming system, the CPU switches automatically from process to process running each for tens or hundreds of milliseconds. In reality, the CPU is actually running one and only one process at a time.

Execution model

Over years, operating system designers evolved a model that makes concurrency easier to deal with. In this model, each runnable software on the computer—often components of the operating system itself—is organized into a number of (sequential) *processes*, each viewed as a block of code with a pointer showing the next instruction to be executed.

How can several processes share one CPU? Operating system takes care of this by making sure:

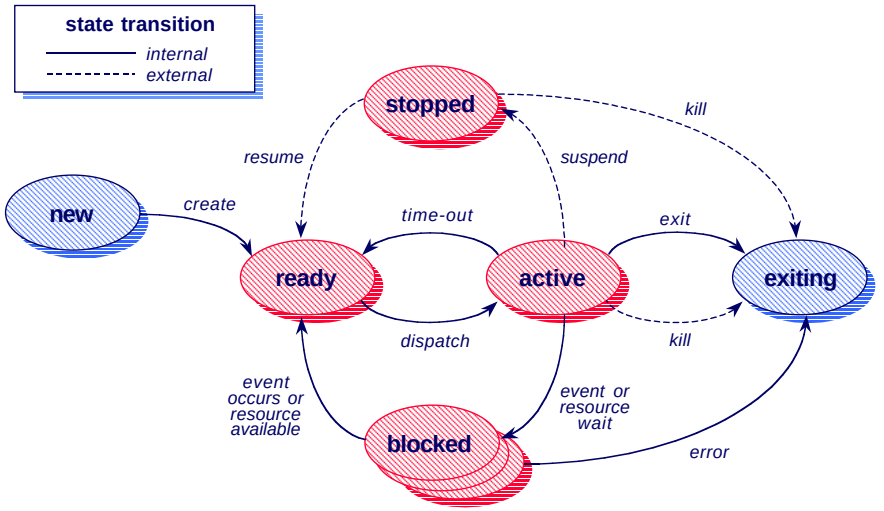
- each process gets a chance to run—*fair scheduling*.
- they do not modify each other's state—*protection*.

Process states

There are a number of *states* that can be attributed to a process: indeed, the operation of a multiprogramming system can be described by a state transition diagram on the process states. The states of a process include:

- **New**—a process being created but not yet included in the pool of executable processes (*resource acquisition*).
- **Ready**—processes that are prepared to execute when given the opportunity.
- **Active**—the process that is currently being executed by the CPU.
- **Blocked**—a process that cannot execute until some event occurs.
- **Stopped**—a special case of **blocked** where the process is suspended by the operator or the user.
- **Exiting**—a process that is about to be removed from the pool of executable processes (*resource release*).

Process state diagram



Process description

The operating system must know specific information about processes in order to manage and control them. Such information is usually grouped into two categories:

- process state information
  - E.g., CPU registers (general purpose and special purpose), program counter.
- process control information
  - E.g., scheduling priority, resources held, access privileges, memory allocated, accounting.

This collection of process information is kept in and access through a *process control block (PCB)*.

Information in both groups are OS dependent.

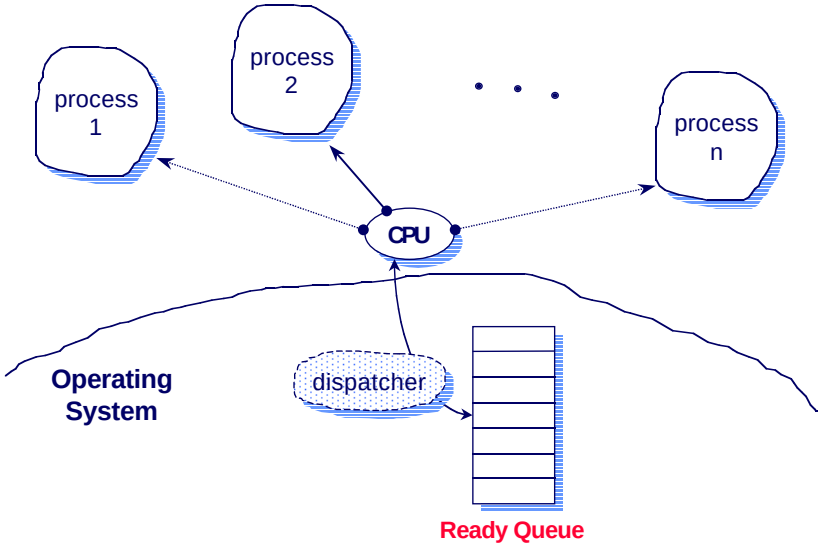
Process scheduling

The objective of multiprogramming is to have some user process running at all times. The OS keeps the CPU busy with productive work by dynamically selecting (scheduling) the next user process to become active.

The (re-)scheduling is performed by a module, called the *dispatcher*. A dispatcher usually only executes the following primitive pseudo-code:

```
loop forever {
        run the process for a while.
        stop process and save its state.
        load state of another process.
    }
```

Dispatcher at work



Control of the CPU

The CPU can only do one thing at a time. While a user process is running, dispatcher cannot run, thus the operating system may lose control.

How does the dispatcher regain control (of the CPU)?

- Trust the process to wake up the dispatcher when done (*sleeping beauty approach*).
- Provide a mechanism to wake up the dispatcher (*alarm clock*).

The problem with the first approach is that sometimes processes loop indefinitely. Therefore, the alarm clock interrupt approach is better.

Context switch

When an event occurs, the operating system saves the state of the *active* process and restores the state of the *interrupt service routine (ISR)*. This mechanism is called a **Context Switch**.

What must get saved? *Everything that the next process could or will damage*. For example:

- Program counter (PC)
- Program status word (PSW)
- CPU registers (general purpose, floating-point)
- File access pointer(s)
- Memory (*perhaps*?)

While saving the state, the operating system should mask (disable) *all* interrupts. Why?

Memory: *to save* or *NOT to save*

Here are the possibilities:

- Save *all* memory onto disk.
  - Could be *very* time-consuming. E.g., assume data transfers to disk at 1MB/sec. How long does saving a 4MB process take?
- Don't save memory; trust next process.
  - This is the approach taken by PCs and MACs.
- Isolate (protect) memory from next process.
  - This is *memory management*, to be covered later.

Context switch implementation

The mechanism of context switching is the most important part of an operating system, and needs special care during the implementation, because:

- It is tricky.

  Saving the state of a user process is problematic because the operating system must execute code to save the state *without* changing the process' *current* state!

- Machine dependent.

  Thanks to technology; each CPU provides some special support to ease the implementation.

Creating a new process

There are two practical ways of creating a new process:

- Build one from scratch:
  - Load *code* and *data* into memory.
  - Create (empty) a *dynamic memory workspace (heap)*.
  - Create and initialize the *process control block*.
  - Make process known to dispatcher.
- Clone an existing one:
  - Stop current process and save its state.
  - Make a copy of *code*, *data*, *dynamic memory workspace* and *process control block*.
  - Make process known to dispatcher.

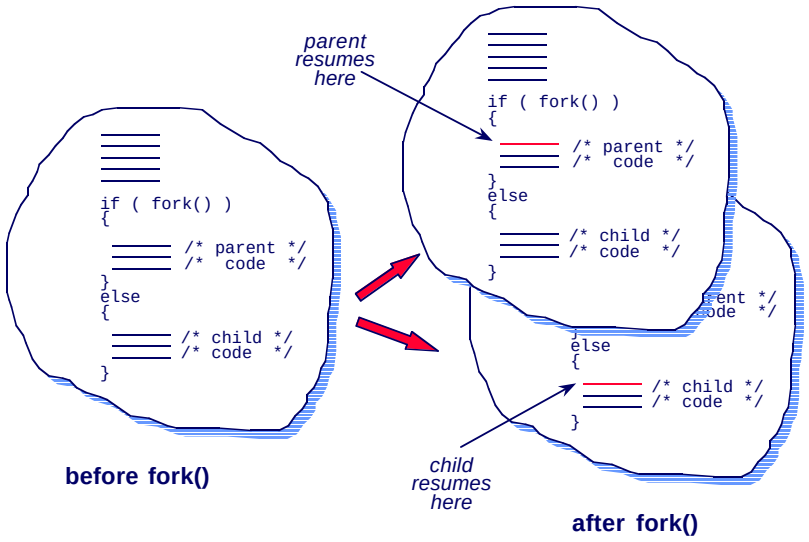
Process creation mechanisms

Who creates the processes and how are they supported? *Every operating system has a mechanism to create processes.*

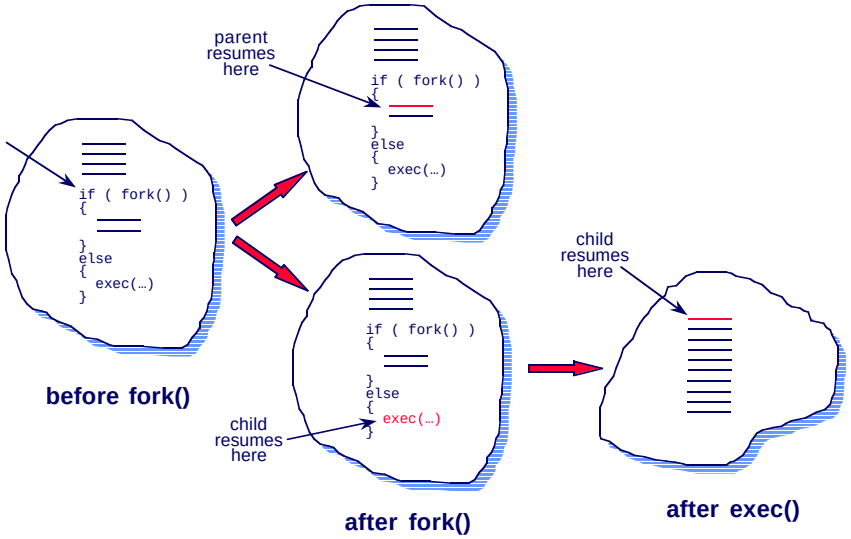
For example, in UNIX the **fork()** system call is used to create processes. **fork()** creates an identical copy of the calling process. After the **fork()**, the *parent* continues running concurrently with its *child* competing equally for the CPU.

On the other hand, in MS-DOS, the **LOAD_AND_EXEC** system call creates a child process. This call suspends the parent until the child has finished execution, so the parent and child do not run concurrently.

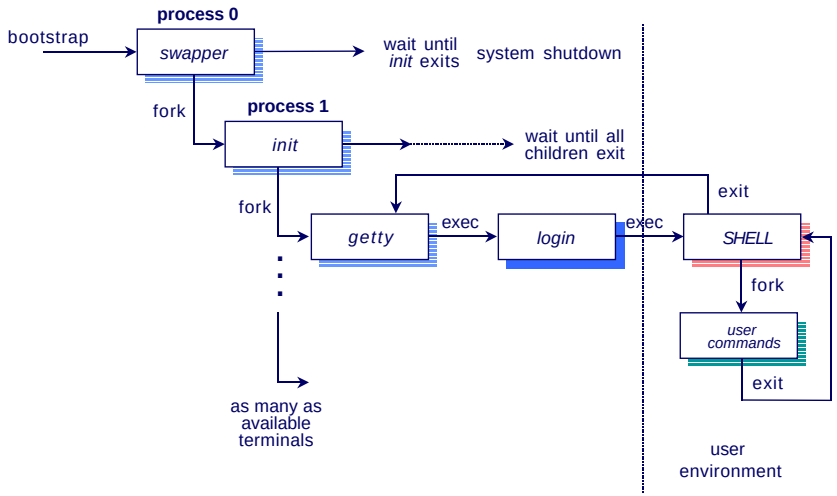
Process creation: UNIX example



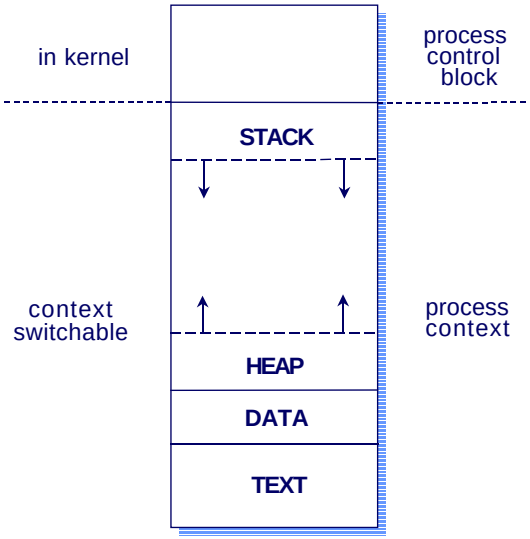
A typical use of **fork()**



UNIX system initialization



A UNIX process context



stack: local variables

heap: dynamic variables

data: constants and static variables

text : executable code

Process termination

A process enters the *exiting* state for one of the following reasons:

- normal completion: A process executes a system call for termination (e.g., in UNIX **exit()** is called).
- abnormal termination:
  - programming errors
    - run time
    - I/O
  - user intervention

Threads

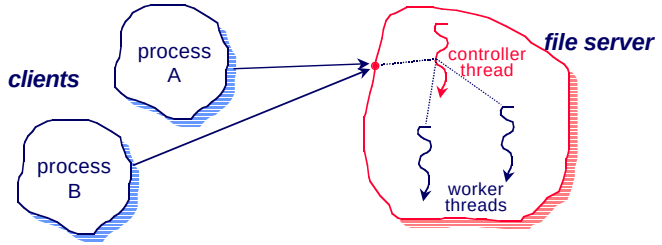
Unit of execution (unit of dispatching) and a collection of resources, with which the unit of execution is associated, characterize the notion of a process.

A *thread* is the abstraction of a unit of execution. It is also referred to as a *light-weight process* (*LWP*).

As a basic unit of CPU utilization, a thread consists of an instruction pointer (also referred to as the PC or instruction counter), a CPU register set and a stack. A thread shares its code and data, as well as system resources and other OS related information, with its peer group (other threads of the same process).

Threads: an example

A good example of an application that could make use of threads is a file server on a local area network (LAN).

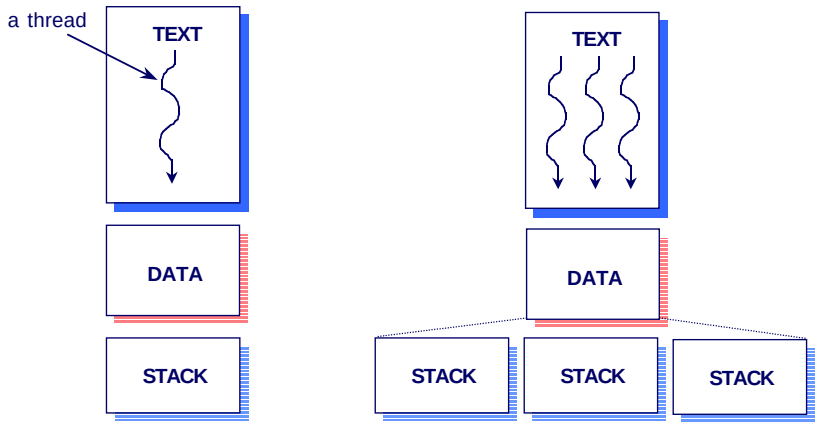


A ‘‘controller’’ thread accepts file service requests and spawns a ‘‘worker’’ thread for each request, therefore may handle many requests concurrently. When a worker thread finishes servicing a request, it is destroyed.

Threads versus processes

- A thread operates in much the same way as a process:
  - can be one of the several states;
  - executes sequentially (within a process and shares the CPU);
  - can issue system calls.
- Creating a thread is less expensive.
- Switching to a thread within a process is cheaper than switching between threads of different processes.
- Threads within a process share resources (including the same memory address space) conveniently and efficiently, unlike separate independent processes.
- Threads within a process are NOT independent and are NOT protected against each other.

Threads versus processes continued



a traditional process **a multi-threaded process (task)**

Thread implementations

- User level:

  implemented as a set of library functions; cannot be scheduled independently; each thread gets partial time quantum of a process; a system call by a thread blocks the entire set of threads of a process; less costly (thread) operations

- Kernel level:

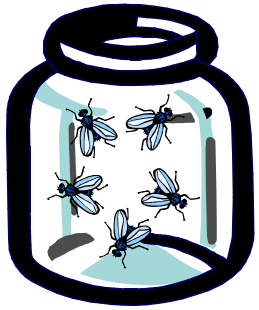
  implemented as system calls; can be scheduled directly by the OS; independent operation of threads in a single process; more expensive (thread) operations.

- Hybrid approach:

  combines the advantages of the above two; e.g., Solaris threads.

Fly in a bottle

A *traditional* UNIX process



A *modern* UNIX process