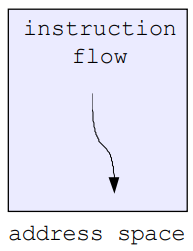
Operating Systems

Processes

What is a “Process”

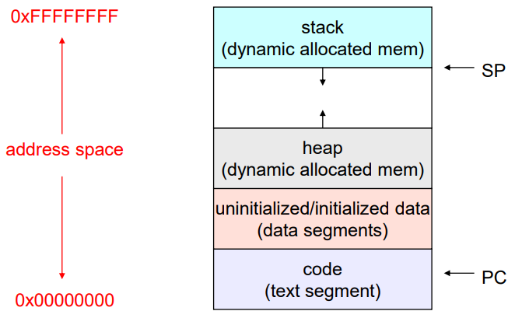
A process is the OS’s abstraction for execution, this is not the same as a program, a program is the list of instructions, initialized data etc… where as a process is a program **in execution**.

A sequential process is a single flow/sequence of instruction in execution (abstraction of the CPU) and an address space (an abstraction of memory).

Only one process can run on a processor core at any instant though different processes may run the same program.

What’s In a Process

A process consists of (at least):

* An address space containing
  + Code (instructions) for the running program
  + Data for the running program (static data, heap data, stack)
* A CPU state, consisting of
  + Program counter (PC), indicating the next instruction
  + Stack pointer, current stack position
  + Other general-purpose register values
* A set of OS resources
  + Open files, network connections, sound channels etc…

In other words, everything needed to run the program (or restart it if interrupted).

The OS Process Namespace

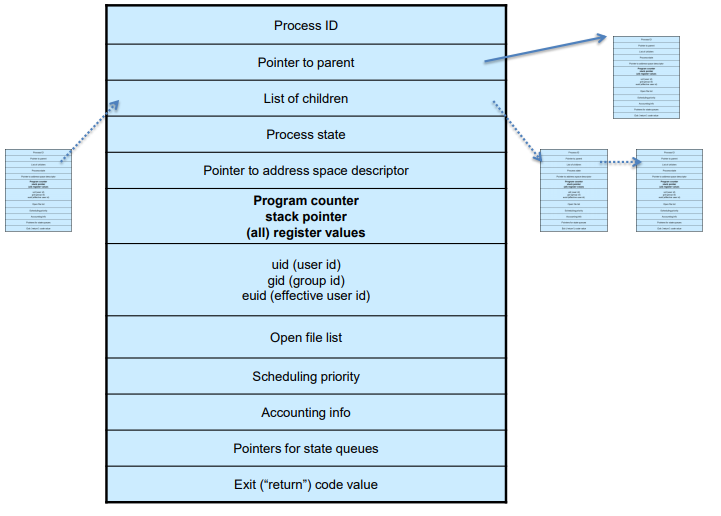
Each process is identified by a process ID (PID) which is an integer. The PID namspaces is global to the system, only one process at a time has a specific PID (there are exceptions to this such as cgroups).

Operations that create processes return a PID (e.g. fork()) and operations on processes take PIDs as arguments (e.g. kill(), wait(), nice()).

Representation of Processes by the OS

The OS maintains a data structure to keep track of a process’s state, this is known as a process control block (PCB) or process/task descriptor and is identified by the PID.

The OS keeps all of a process’s execution state (PC, SP, registers etc..) in (or linked from) the PCB when the process isn’t running (when a process execution is stopped, its state is transferred out of the hardware into a PCB). When the process is running its state is spread between the PCB and the hardware (CPU registers).

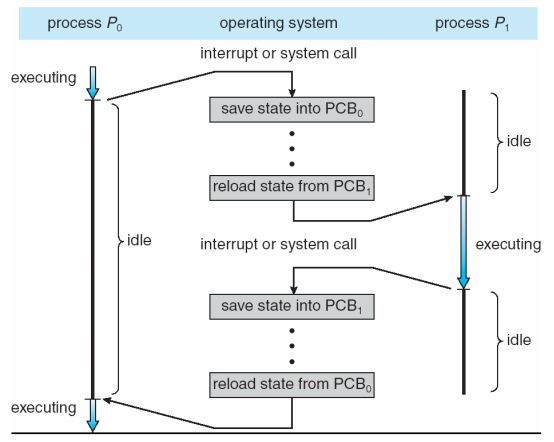
The PCB

The PCB is a data structure with many, many fields:

* Process ID (PID)
* Parent process ID
* Execution state
* Program counter, stack pointer, registers
* Address space info
* UNIX user id, group id
* Scheduling info
* Accounting info
* Pointers for state queues

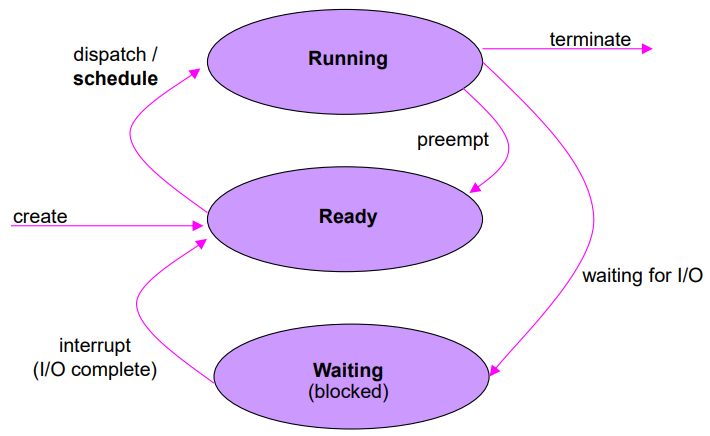
In linux the PCB is known as a task\_struct and is defined in (include/linux/sched.h) with more than 100 fields.

PCBs and CPU State

When a process is running, its CPU state is inside the CPU. When the OS gets control because of a syscall, exception or interrupt the OS saves the CPU state of the running process in that process’s PCB.

When the OS returns the process to the running state it loads the hardware registers with values from that process’s PCB (the general purpose registers, stack pointer and instruction pointer, plus any extra the hardware may have).

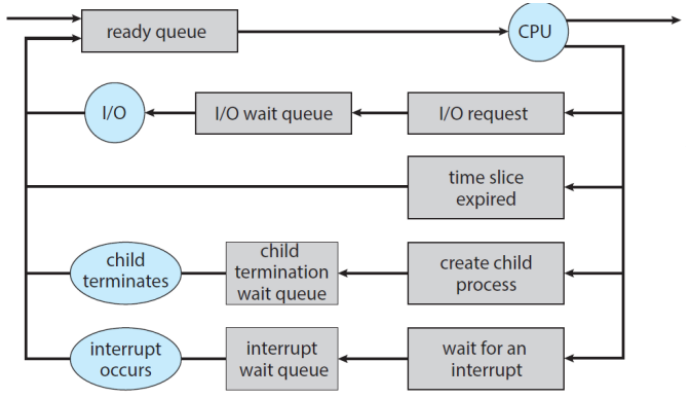
The act of switching the CPU from one process to another is called a context switch, systems may do this 100s or 1000s times a second, this takes a few microseconds on today’s hardware, this is expensive relative to thread-based context switches. Choosing which process to run next is called scheduling.

Process Execution States

Each process has an execution state which indicates what it’s currently doing:

* Ready: waiting to be assigned to a CPU, could run but another process has the CPU
* Running: executing on a CPU
* Waiting (aka ‘blocked’): waiting for an event such as I/O completion or a message from another process, it cannot make progress until the event happens.

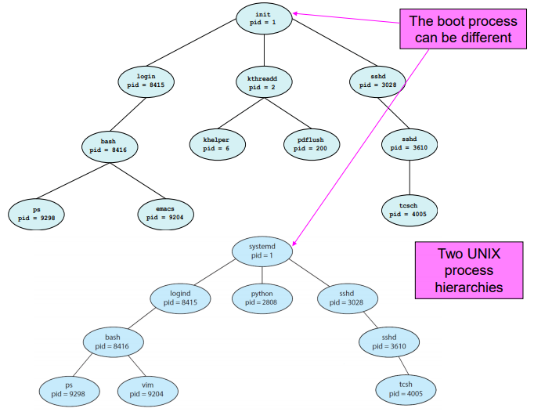
As a process executes, it moves from state to state.

State Queues

The OS maintains a collection of queues that represent the state of all processes in the system. There is typically one queue for each state but there may be multiple waiting queues. Each PCB is queued onto a state queue according to the current state of the process it represents, as a process changes state, its PCB is unlinked from one queue and linked onto another. These lists are likely implemented as linked lists (doubly) to make moving PCBs between queues easier.

There may be one wait queue for each type of wait (specific device, timer, message,…).

PCBs and State Queues

PCBs are data structures that are dynamically allocated inside OS memory. When a process is created the OS allocates a PCB for it, initializes that PCB (does things unrelated to the PCB) and puts that PCB on the correct queue. As the process computes the OS moves its PCB from queue to queue.

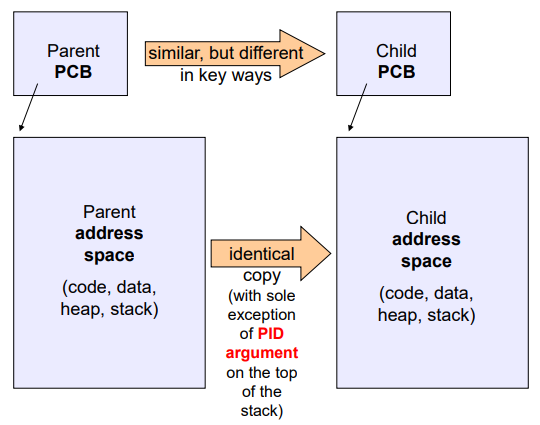
When a process is terminated the PCB may be retained for a while (to receive signals etc…), eventually the OS will deallocate the PCB.

Process Creation

New processes are created by existing processes, the creator is called the parent and the created process is called the child.

Depending on the OS the child processes inherit certain attributes of the parent (such as open file tables). On some systems resources allocated to parent processes may be divided among its children.

(In UNIX) when a child is created the parent may either wait for the child to finish, or continue in parallel.

UNIX Process Creation Details

UNIX process creation is achieved through the fork() system call. This creates and initializes a new PCB providing the child resources that belonged to the parent and initializes the PC and SP to be the same as the parent’s. This also creates a new address space initialized with a copy of the entire contents of the address space of the parent and finally places the new PCB on the ready queue.

The fork() system call ‘returns twice’, once into the parent, returning the child’s PID, and once to the child, returning 0.

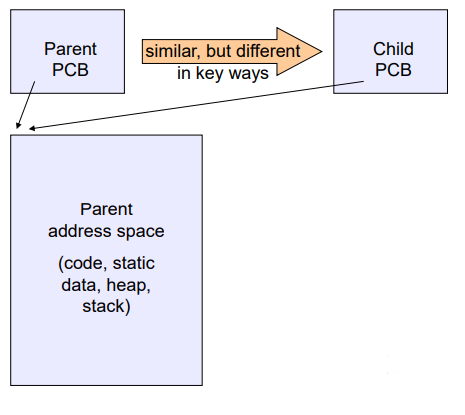
Essentially in UNIX, fork() means ‘clone me’.

exec() Vs fork()

So how do we start a new program instead of just forking the old program? First we fork, then we use exec() ( int exec(char \* prog, char\* argv[]) ). exec() stops the current process, loads ‘prog’ into the address space over-writing the existing process image, initializes the hardware context with args for the new program and places the PCB onto the ready queue. Note that this didn’t create a new process, just replaced the calling process.

Making Process Creation Faster

The semantics of fork() say the child’s address space is a copy of the parent’s but implementing fork() that way is **slow**, we have to allocate memory for the new address space, set up the child’s page tables to map a new address space and copy the parent’s address space contents which we are likely to destroy with an exec().

Method 1: vfork()

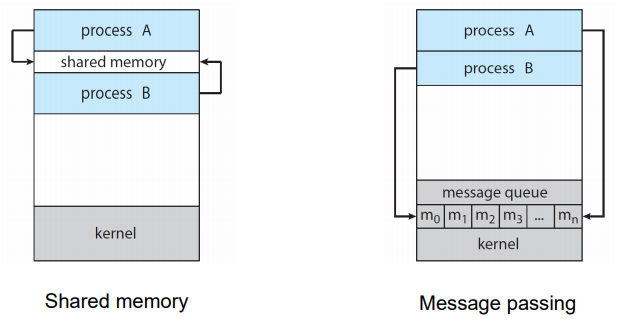
vfork() is the older (now uncommon) method of the two we’ll discuss. Instead of “child’s address space is a copy of the parent’s” the semantics are “ child’s address space **is** the parent’s” with a promise that the child won’t modify the address space before doing and execve() (this is unenforced, so use vfork() at your won risk). When execve() is called a new address space is created and it’s loaded with the new executable. This method saves the wasted effort of duplicating the parent’s address but the parent is blocked until execve() is executed by the child.

Method 2: Copy-On-Write (COW)

This method retains the original semantics but copies “only what is necessary” rather than the entire address space. On fork() a new address space is created, the page tables are initialized with the same mappings as the parents (they both point to the same physical memory), at this point no copying has occurred (except the top page of the stack). Both the parent and child page tables are set to make all pages read-only, if either writes to memory an exception occurs. When the exception occurs the OS copies the page and adjusts the page tables, etc…

Inter-process Communication

Processes can be independent, meaning they don’t talk or work together, they’re completely separate, or they could cooperate, which means they work together making use of:

* Information sharing, more applications interested in the same information
* Computation speedup, to exploit parallel hardware
* Modularity, reusability of components

Cooperating processes require inter-process communication (IPC) mechanisms to send and receive data, there are two main mechanism, shared memory and message passing.

Shared Memory

We allow processes to communicate and synchronize by sharing a part of their address space. The OS doesn’t mediate communication in this address space meaning there’s no overhead as the OS usually prevents processes from accessing each others memory. This requires processes to agree (have permission) to void this restriction.

The format of the data in this shared area needs to be decided by the application and the application programmer has to fully manage the data transfer which is **not trivial**.

This method tends to be used for sending large (single) objects such as images, tables, etc…

Message Passing

We allow processes to communicate and synchronize without sharing part of their address space while having the OS mediate the communication (which may introduce overhead).

Message passing allows communication between processes on the same machine and among different internetworked machines which isn’t possible with shared memory.

The message-passing facility provides at least two operations, send(message) and receive(message).

Message passing has multiple implementation tradeoffs for example message sizes which could be fixed or variable, each with their own drawbacks.

Message Passing – Naming

Communicating processes must refer to each other (this is also a problem in shared memory). Communication can be done directly in two ways:

* Symmetric: Explicit name of sender and receiver
  + send(P, message) – sends a message to process P
  + receive(Q, message) – receives a message from process Q
* Asymmetric: Explicit at at-least one end
  + send(p, message) – sends a message to process P
  + receive(id, message) – receive from any process and save the sender in id

Communication could also be done indirectly, here there’s no need to know the sender and/or receiver in advance. This is achieved using mailboxes (e.g. POSIX Mailbox) and functions send(A, message) which sends a message to mailbox A and receive(A, message) which gets a message from mailbox A. Mailboxes can be accessed by more than two process and two processes can use multiple mailboxes to communicate.

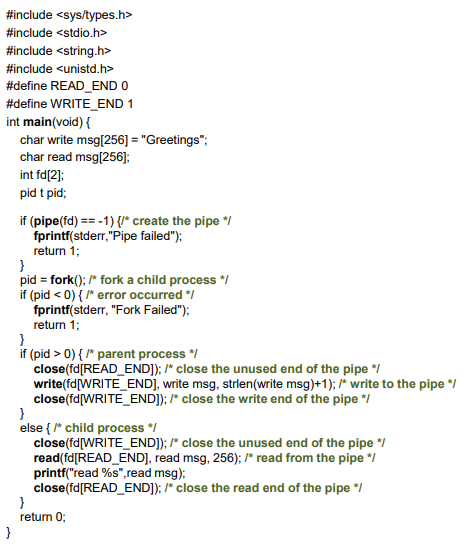
Message Passing – Synchronization

There are also different design options for implementing send() and receive(), they could be blocking or nonblocking and synchronous or asynchronous, 4 different combinations of these plus send and receive don’t need to be the same making 8 different combinations.

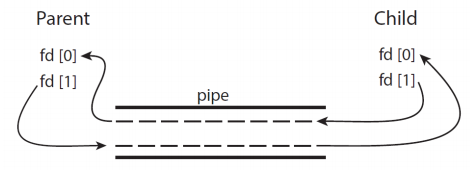
Both send() and receive() being blocking is known as a rendezvous and is a simple solution to the consumer-producer problem.

Message Passing – Buffering

Messages exchanged reside in temporary buffer/queues, these can be:

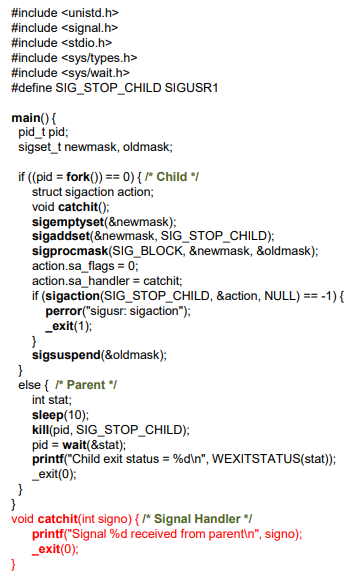
* Zero capacity (no buffering), the mailbox won’t store messages, the sender must block until the recipient receives the message.
* Bounded capacity, the buffer/queue is a finite length n (can only hold n messages), if the buffer/queue isn’t full it will store new messages however if it is, senders must again block.
* Unbounded capacity, there is an infinite queue and senders never have to block.

Message Passing Example: Pipes

A pipe acts as a conduit allowing two processes to communicate (with one way data flow). Pipes can be anonymous (ordinary) between a parent and child or named and between any pair of processes. In UNIX pipes are known as FIFO.

Client-Server Communication

Sockets are a common abstraction, they’re endpoints for communications and are identified by an IP address concatenated with a port number. Servers implementing specific serices (SSH, FTP, and HTTP) listen to well-known ports.

Remote procedure call (RPC) is an abstraction of the procedure-call mechanism for use between systems with network connections. This is similar to IPC in many respects and uses message-based communication to provide remote services.

Signals

Signals are an OS mechanism to notify a process (one way) and don’t carry information themselves. Signals can be thought of as a software-generated interrupt/exception that can be synchronous meaning caused by the process (e.g. division by zero error) or asynchronous meaning not (e.g. timer/alarm). Think of signals as the revers syscalls, syscall: process -> OS, signal: OS -> process.

Signals can also be sent from one process to another (or itself), again this is only a notification, no data is communicated. This can be used for management to kill a process, synchronization etc..

Signal handlers are functions that process a signal and do not return a value, these handlers must be registered or the OS will use its default action. Signals can be sent with kill() or raise().