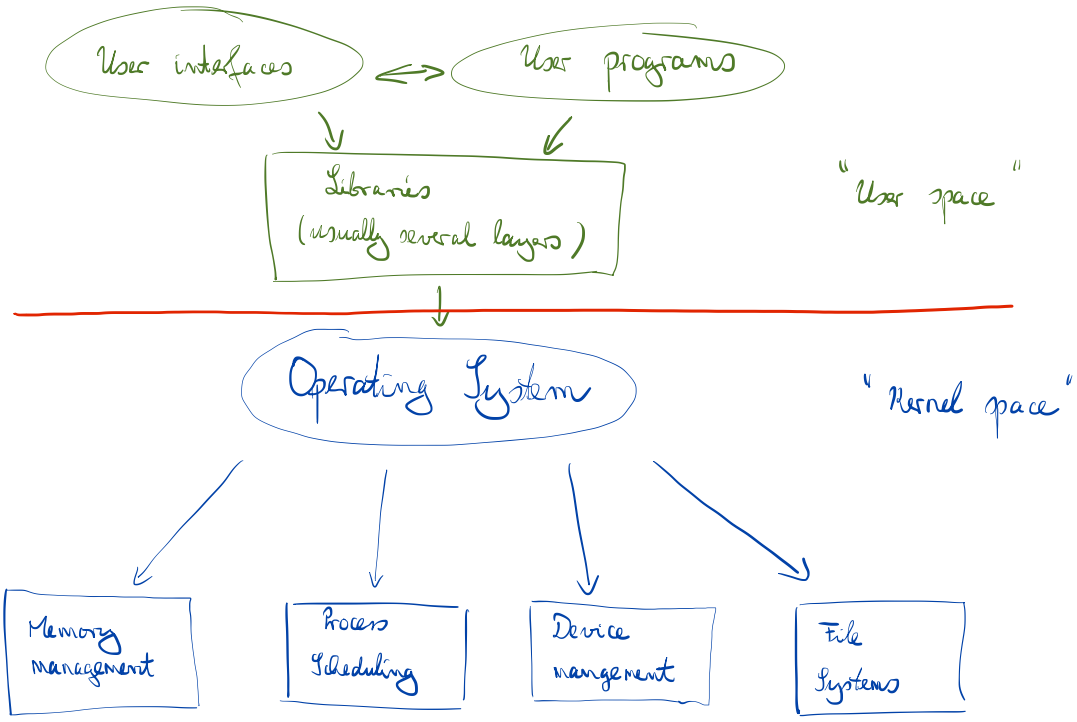
Operating Systems

Privileged program(s) running on a computer to

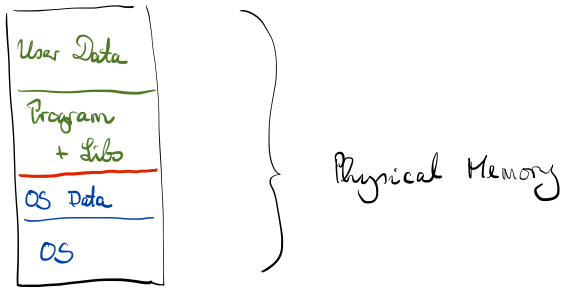
- provide abstract interfaces to hardware
- manage resources



Let's look at these functions in turn.

Memory management:

① Primitive OSs: "Monoprogramming" (today only simple embedded systems)

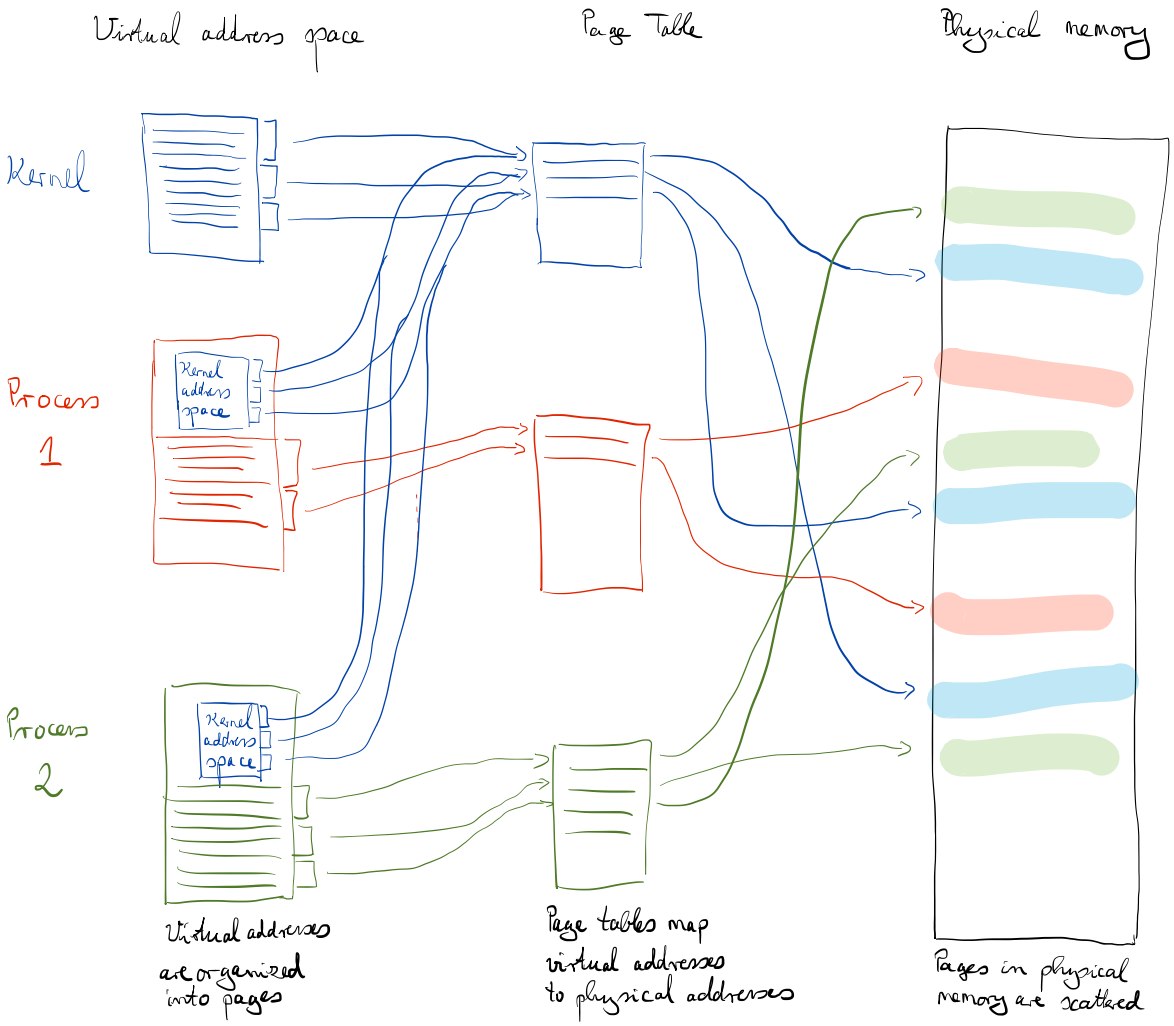


Only one program can run at a time and uses (potentially) all memory.

② Partitioning: Partition memory into several chunks, one for each program.

- Still very primitive

③ Virtual memory



- Memory is divided into pages of fixed size (usually 4K or 64K)
- Every process sees a linear address space (at ISA level, so hardware)
- Virtual pages are mapped to physical pages via the page table
  - caching via "TLB" — Translation Lookaside Buffer
  - all in hardware (fast!)
  - 1 table per process + one for the kernel (protected and mapped into process address space to enable userspace-kernel communication)
- Even assembly code "sees" only the virtual addresses (except for kernel mode)
- Only used virtual address space is physically allocated
- Processes can share physical pages ("COW": copy-on-write)
- Pages can be "swapped" out to disk storage ("demand paging")

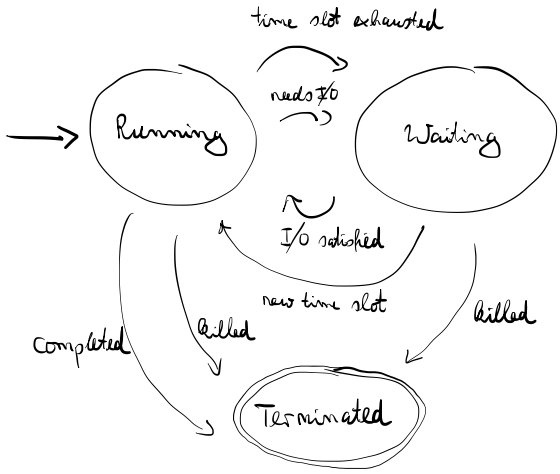
Remark: "Demand segmentation" is an older concept from the CISC era, still available on x86 CPUs, but not used by modern operating systems.

Process management

Terminology:

- Program : set of instructions (on disk or in memory)
- Job : The act of running a program, which may involve loading a program into executable memory and spawning one or more processes
- Process : A program in execution, with its own virtual memory area, TLB, program counter, stack, ...

Process states:



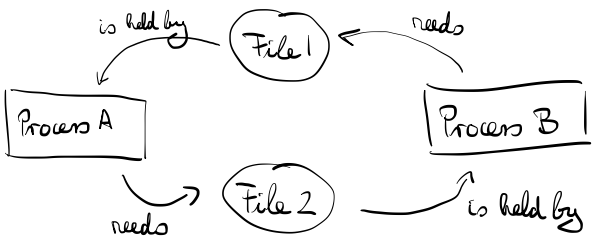
Process Scheduler:

- Manage time slots (issue: latency vs. throughput)
- Manage I/O requests
- Assign processes to CPUs

Requirements:

- Fairness
- Latency
- Throughput
- Avoid deadlock and starvation (within the OS and at the interface to user space, multi-process (multithreaded) applications need to take care of this themselves!)

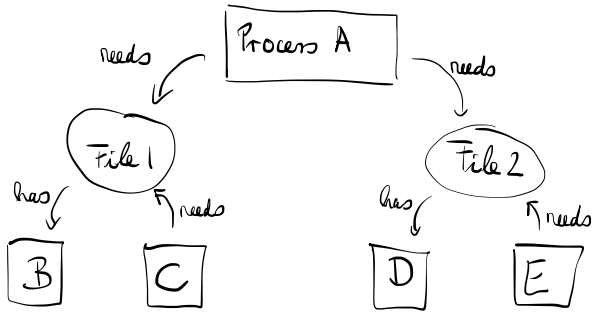
Deadlock: circular waiting for resources, e.g.



necessary conditions for deadlock:

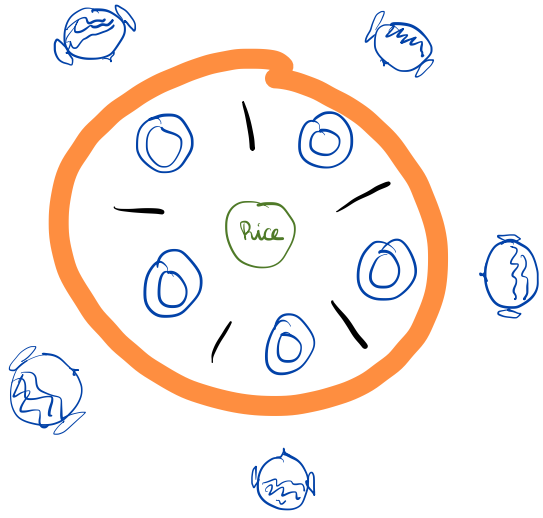
- allocation of resources is exclusive
- a process can hold a resource while waiting for another resource
- no preemption (the OS cannot temporarily reallocate a resource)
- circular waiting

Starvation: Some process can never get all resources it needs



If A only runs if it has File 1 and File 2, B and C can alternate holding and requesting File 1, likewise D and E for File 2, without A ever getting the chance to proceed. We say A is starved.

The "dining philosopher's problem" as a paradigm for deadlock and starvation



- 5 philosophers sit around a table
- each one either
  THINKS,
  is HUNGRY,
  or EATS
- To eat, needs two chopsticks
- There is only one between each pair of philosophers

The problem: Simple solutions do not avoid deadlock/starvation, e.g.

- THINK for an arbitrary amount of time until HUNGRY
- wait until left chopstick is available; when it is, pick it up
- wait " right " " " " ; " " " " " "
- when both are held, EAT for a bounded amount of time
- relase chopsticks, repeat

What can go wrong?

Resource hierarchy solution:

- Number the chopsticks 1, ..., 5
- Each philosopher must pick up the lower-number chopstick first, wait otherwise.

→ avoids deadlock (why?)

→ not fair, if one philosopher is slow, can starve. (How?)

Arbitrator solution:

- Philosophers must ask waiter if they may eat
- Waiter will let only one philosopher eat at a time

→ low concurrency

→ fairness can be implemented as policy for the waiter

Limit number of diners at the table:

- Let only n-1 philosophers sit down
- Whoever finishes eating gets up and lets the other sit down

Dijkstra/Tannenbaum solution:

→ Complicated, see (Pseudo-) Code on Wikipedia

Chandy/Misra solution:

- each chopstick is always in the possession of one of its two philosophers
- chopsticks can be DIRTY or CLEAN
- INIT: Make sure one philosopher has two chopsticks,
  all chopsticks are DIRTY
- when HUNGRY: request chopsticks not already held
- when THINKING or HUNGRY: honor requests for DIRTY chopsticks
  CLEAN them before passing
  DEFER requests for CLEAN chopsticks
- EAT for bounded time when all chopsticks are held:
  DEFER all requests
  chopsticks become DIRTY
- After EATing: honor all DEFERred requests (again CLEANing before passing)

Proof & correctness:

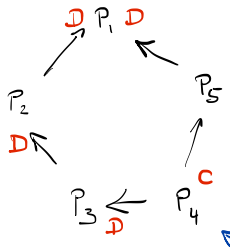
- Philosophers are vertices of a directed graph G
- $U \to V$ ("U has precedence over V") if
  - U holds shared chopstick, and it is CLEAN
  - V " " " , and it is DIRTY
- $U \to V$ flips only iff U starts eating (dirtying the chopstick)
- an EATing philosopher has only incoming edges

Lemma: G is always acyclic

Proof: G is initially acyclic. A change occurs only when a philosopher U eating, but then all edges are incoming, so U cannot become part of a cycle. □

Theorem: Every hungry philosopher will eat.

Idea:



↖ highest priority, will eat before P_5 and either before P_3 or after P_3 has finished.

Proof: Suppose u is HUNGRY

v is a neighor, holding the chopstick some time before u gets to eat

(i) the chopstick v holds is DIRTY

v will hand it over immediately or after finishing to eat

⇒ u will then hold a CLEAN chopstick and not give it up before eating

(ii) the chopstick v holds is CLEAN

If v gets to EAT, the chopstick will become DIRTY and (i) applies

So, u has to wait until the neighbor with higher priority gets to eat once.

Since there is a philosopher with highest priority since G is acyclic, this happens in finite time. □

Advantages of the Chandy/Misra approach:

- Generalizes to arbitrary networks
- Symmetric (no philosopher is special)
- Does not deny any feasible system state (→ concurrency)

Note that this solution requires to pass simple messages!