**Processor** controls hardware and executes instructions.
**Memory** stores data and code.
**I/O components** read and write from I/O devices.
**System bus** interconnects different components and provides communication between them.

OS must make **efficient** use of available resources and must **share** resources (CPU cores, caches, RAM, GPUs, clocks, timers, persistent storage) among multiple users.

**Providing clean interfaces** allows us to convert raw hardware into a usable computer system by keeping **details** hidden from user-level software.

Must support several simultaneous parallel activities, and may switch activities at arbitrary times, so must ensure **safe concurrency**.

**MULTIPROCESSORS** (Windows, macOS, Linux) have many CPU cores and CPUs.

**SERVER OS** (Solaris, FreeBSD, Linux) shares hardware/software resources.
**MAINFRAMES** have bespoke hardware, limited workload.
**SMARTPHONES** have power-efficient CPUs.
**EMBEDDED OSs** (QNX, VXWorks) are used for home utilities, and only run trusted software.
**REAL-TIME OSs** are time-oriented (not performance/IO).
**SENSOR NETWORK OS** (Tiny OS) is resource efficient.

The OS **kernel** is a privileged program that implements OS functionality.

**Monolithic** kernels are a single black box with all functionality. One executable with its own address space. Easier to write kernel components but overall design is complex, with no protection between them.

**Microkernels** have as little functionality in the kernel as possible, functionality passed to a user-level server. Kernel does IPC between servers - separate ones for I/O, file access, scheduling, but this leads to high overhead. Less error-prone and servers have clean interfaces.

**Hybrid** kernels combine features of both.

**LINUX** is a variant of Unix. **Interrupt handlers** are the primary means to interact with devices. I/O scheduler orders disk operations. Supports static in-kernel components and dynamically loadable **modules**.

NTOS provides **system calls** in **WINDOWS**. Programs are built on top of **dynamic code libraries**. NTOS loaded from *ntoskrnl.exe* at boot, consists of **executive** and **kernel** layers. **Device drivers** loaded into kernel memory. NT structured like a **microkernel**, most components run in same address space. **Hardware Abstraction Layer** abstracts out BIOS config, CPU types, etc...

# PROCESSES

allow a single processor to run multiple programs "**simultaneously**". Provides concurrency, isolation between programs, allows better resource utilisation, simplifies programming.

**Pseudo concurrency** — single physical processor is switched between processes by interleaving.

**Real concurrency** utilises multiple physical processors.

On a **context switch**, the processor switches from executing process A to executing process B. May occur in response to events/interrupts, or as a result of periodic scheduling. Process may be **restarted later**, so all information needed should be stored in **process descriptor** / **control block**, kept in the **process table**.

**Process Control Block** — process has its own VM, should store PC, page table register, stack pointer, ..., process management info, file management info.

Context switches are expensive, so avoid unnecessary usage. *Direct cost* — save/restore process state. *Indirect cost* — perturbation of memory caches.
^ Flushing TLB & pipeline flushing

UNIX allows processes to create hierarchies. Windows has no notion of this.
***int fork(void)*** — Creates new child process, to be executed concurrently, by making exact copy (same resources) of parent process image. Returns child PID in **parent** process and 0 in **child** process. No child created, and -1 returned to parent on error.
***int execve(char *path, char *argv[], char *envp[])*** — Changes process image and runs new process. Lots of useful wrappers.
***int waitpid(int pid, int* status, int options)*** — Suspends execution of calling process until process *pid* terminates normally or signal received. Can wait for more than one child.

**Simple** design philosophy, so process API made up of basic blocks that can be easily combined.

*CreateProcess()* (equivalent of *fork()* & *execve()*) has 10 parameters!

**Termination** — normally process completes execution of body, calls *exit()*. **Abnormal exit** — runs into error or unhandled exception. **Aborted** — another process has overruled execution. Some run in an endless loop.

***void exit(int status)*** — Terminates a process. Never returns in the calling process.
***int kill(int pid, int sig)*** — sends signal sig to process pid.

| | |
|---|---|
| SIGINT | Interrupt from keyboard |
| SIGABRT | Abort signal from abort |
| SIGFPE | Floating point exception |
| SIGKILL | Kill signal |
| SIGSEGV | Invalid memory reference |
| SIGPIPE | Broken pipe: write to pipe with no readers |
| SIGALRM | Timer signal from alarm |
| SIGTERM | Termination signal |

**SIGNALS** are an IPC mechanism — **delivery** similar to delivery of hardware interrupts. Process can send signal to another process, if it has permission. Generated when an exception occurs, when kernel wants to notify process of an event, when certain key combinations typed in terminal, or programmatically using *int kill()*.

Default action for most signals is to terminate process, but receiving process may ignore/handle it instead (unless *SIGKILL* or *SIGSTOP*).

**PIPES** — method of connecting stdout of one process to stdin of another. **Named** or **unnamed**. Sender should close the read end, receiver should close the write end.

**Named Pipes** / **FIFOs** are persistent pipes that outlive processes which created them. Stored on file system. Any process can open it like a regular file.

***int pipe(int fd[2])*** — returns two file descriptors: read end *fd[0]*, write end *fd[1]*.

# THREADS

are execution streams that share the **same address space**. **Multithreading** — each process can contain multiple threads.

Useful for applications which contain multiple parallel activities that read/write the same data. Processes are too heavyweight — it is difficult to communicate between address spaces, and expensive to context switch between them & create/destroy activities.

Shared address space can lead to memory corruption and concurrency bugs. How to handle **forking** and **signals**?

**PTHREADS** implemented by most Unix systems.
*int pthread_create(...)* — Creates a new thread.
***void pthread_exit(void value_ptr)*** — Terminates the thread and makes *value_ptr* available to any successful join with terminating thread. Called implicitly when the thread's start routine returns.
*int pthread_yield(void)* — Releases CPU to let another thread run. Returns 0 on success (always), or error code.
*int pthread_join(...)* — Blocks until thread terminates.

**User-level threads** — OS kernel not aware of threads, each process manages its own threads. Better performance, and each application can have its own scheduling algorithm. But blocking system calls (e.g. during page fault) stop **all threads** in process.

**Kernel-level threads** — Blocking syscalls / page faults can be easily accommodated. But thread operations are expensive. No application-specific schedulers.

**Hybrid** approaches use kernel threads and multiplex user-level threads onto some of these.

# SCHEDULING — if multiple processes are ready, which one should be run?


**Goals of Scheduling Algorithms** => **Ensure fairness**: Comparable processes should get comparable services; **Avoid indefinite postponement**: No process should starve; **Enforce policy**: E.g. priorities; **Maximize resource utilization**: CPU, I/O devices; **Minimize overhead**: From context switches, scheduling decisions
**For Batch Systems** => **Throughput**: Jobs per unit of time; **Turnaround time**: Time between job submission & completion
**For Interactive Systems** => **Response time**: Time between request issued and first response

**Non-preemptive** scheduling lets process run until it blocks / releases CPU. **Preemptive** scheduling lets process run for a maximum amount of fixed time.

**FIRST-COME FIRST-SERVE** — No indefinite postponement. Easy implementation. But suffers from "head-of-line blocking" if long job followed by many short jobs.

**ROUND-ROBIN** — Process runs until it blocks or time quantum exceeded. Fair, and good response time. Poor turnaround time when jobs have similar run-times.
^ Good avg. turnaround time when different runtimes.
***RR quantum (time slice)*** => should be larger than context switch cost, but still provide decent response time; ***RR overhead*** => RR quantum/(RR quantum+ctx. switch time).

Pick the **SHORTEST JOB FIRST** with run-times known in advance. Optimal when all jobs available simultaneously.

Pick the process with **SHORTEST REMAINING TIME**. Allows new short jobs to get good service.

Run-times usually not available in advance. Compute CPU
**FAIR-SHARE SCHEDULING** — Users assigned some fraction of the CPU.
^ within each user can use RR to schedule.

**PRIORITY SCHEDULING** — Jobs run based on their priority. Can be **externally-defined**, based on some **process-specific metrics**. Can be static or dynamic.

Generally, favour short and I/O-bound jobs. Quickly determine the nature of job and adapt to changes.

**MULTILEVEL FEEDBACK QUEUES** — Form of priority scheduling. One queue for each priority level. Within each queue, can use diff. scheduling (usually RR). Need to determine current nature of job, worry about starvation of lower-priority jobs. *Feedback* comes from periodically recomputing priorities => e.g. based on how much CPU they used recently, increase job's priority as it ages, exponentially weighed moving average. Not very flexible, does not react quickly to changes, cheating is a concern.

**LOTTERY SCHEDULING** — Jobs receive lottery tickets for various resources — random ticket chosen at each scheduling decision. Number of lottery tickets is meaningful as a result. Highly responsive with no starvation. Jobs can exchange tickets. But unpredictable response time.

# How do processes SYNCHRONISE their operations to perform a task?

**Critical section** — processes access a shared resource.
**Synchronisation mechanism** required at entry/exit of critical section.
**Race condition** occurs when multiple threads read and write shared data within critical section => their execution interleaved non-deterministically.

Assume **sequential consistency** throughout — operations of each thread appear in program order, operations of all threads executed in sequential order atomically.

**Synchronisation mechanism** required at entry/exit of critical section => mediate concurrent access & prevent race conditions.
An **atomic operation** is a sequence of 1+ statements that are/appear to be indivisible.
**TEST & SET LOCK** is an atomic instruction TSL(*L*) provided by most CPUs. Sets memory location *L* to 1 and returns old value; locks using busy waiting (e.g. while (TSL(*L*) != 0) /*wait*/;) are **spin locks**.
**DISABLING INTERRUPTS** only works on single-processor systems. Misbehaving/buggy processes may never release the CPU.
**STRICT ALTERNATION** fails to meet rules of mutual exclusion => if $T_0$'s non-crit. section is long enough, it will prevent $T_1$ from entering crit. section. ⇓⇓⇓

$T_0$ — turn: 0 — $T_1$

```
while (true) {
    while (turn != 0)
      /* loop */ ;
    critical_section()
    turn = 1;

noncritical_section0();
}
```

```
while (true) {
    while (turn != 1)
      /* loop */ ;
    critical_section()
    turn = 0;

noncritical_section1();
}
```

**PETERSON'S SOLUTION** fixes issues w/ strict alternation; thread $T_k$ calls enter/leave_critical(*k*); ⇓⇓

```
int turn = 0;
int interested[2] = {0, 0};
```

```
void leave_critical(int thread)
{
  interested[thread] = 0;
}
```

```
// thread is 0 or 1
void enter_critical(int thread)
{
  int other = 1 - thread;
  interested[thread] = 1;
  turn = other;
  while (turn == other &&
         interested[other])
    /* loop */ ;
}
```

**Busy waiting** => synch. mechanism for which the waiting thread remains in running state *(e.g. spin locks)* which wastes CPU cycles, and runs into **Priority inversion Problem** => if higher-priority *H* is busy-waiting on lock held by lower-priority *L*, then *H* is always scheduled meaning *L* cannot make progress, i.e. neither can make progress.

**Mutual exclusion** ensures if a process executes its critical section, no other process can be executing it.
1. No two processes may be simultaneously inside a critical section.
2. No process running outside critical section may prevent other processes entering the critical section.
3. No process requiring access to its critical section can be delayed forever.
4. No assumptions made about the relative speed of processes.

^ a commonly desired feature of synchronisation mechanisms.
**Example implementations** => disabling interrupts, strict alternation, Peterson's solution, spin locks, binary semaphores w/ init. *counter*=1.

**LOCK/MUTEX** => mutual exclusion mechanism owned by actively excluding thread; **lock** is usually user/process object while **mutex** is same thing but OS/Kernel object.
**Lock granularity** => the amount of data a lock is protecting.
**Lock overhead** => cost of using locks, e.g. memory space, initialization, time required to acquire/release locks.
**Lock contention** => number of processes waiting for lock; more contention => less parallelism.
**Read/Write Locks** can be used to minimize lock contention & maximize concurrency => exclusive access in write mode but multiple threads can acquire in read mode.

**SEMAPHORES** — processes cooperate by means of signals. Special variables, accessible via (atomic) *down(s), up(s), init(s, i)*. Two private components — a **counter** and a **queue** of processes waiting on the semaphore.
In a **binary** semaphore, counter initialised to −1.

```
init(s, i) ::= counter(s) = i
               queue(s) = {}
```

```
down(s) ::= if counter(s) > 0
              counter(s) = counter(s) - 1
            else
              add P to queue(s)
              suspend current process P
```

```
up(s) ::= if queue(s) not empty
            resume one process in queue(s)
          else
            counter(s) = counter(s) + 1
```

**MONITORS on COND-VARS:**
**Monitors** regulate access to shared data & consist of (implicit) ***monitor lock*** and ≥ 1 ***cond-vars***; ***entry*** procedures called from outside the monitor, ***internal*** procedures only callable from monitor procedures, processes can only call entry procedures — only one process can be in the monitor at one time
**Cond-vars** associated with high-level conditions; if ***cond-var*** signalled w/ no one waiting for it then the signal is lost; has operations:
- *wait(c)* => release monitor lock & wait for *c* to be signalled
- *signal(c)* => wake up one process waiting for *c*
- *broadcast(c)* => wake up all processes waiting for *c*

**Monitors** are a *language construct* and *not supported by C (supported by PintOS via explicit monitor locks)*.
Java supports variant of monitors via **synchronized** methods, but they lack condition variables, i.e. no wait()/notify()

**Producer/consumer example w/ monitors** ⇓⇓

```
monitor ProducerConsumer
    condition not_full, not_empty;
    integer count = 0;

    entry procedure insert(item)
       while (count == N) wait(not_full);
       insert_item(item); count++;
       signal(not_empty);

    entry procedure remove(item)
       while (count == 0) wait(not_empty);
       remove_item(item); count--;
       signal(not_full);
end monitor
```

^ producer inserts & consumer removes items w/ max. capacity *N* => only insert/remove if mutual exclusion achieved => only insert if non-full & remove if non-empty.