- `fork()` creates a new process (child) by duplicating the calling process.
- The child process receives a copy-on-write duplicate of the parent's memory and file descriptor table.
- Use `wait()` or `waitpid()` in the parent to reap the child and avoid zombie processes.
- Each child should call `exit()` (or return from `main()`) to terminate cleanly.
- All `exec*()` functions replace the current process image with a new program.
- `execl(path, arg0, ..., NULL)` — List of arguments; full path required.
- `execlp(file, arg0, ..., NULL)` — List + `PATH` lookup.
- `execle(path, arg0, ..., NULL, envp)` — List + custom environment.
- `execv(path, argv[])` — Vector of arguments; full path.
- `execvp(file, argv[])` — Vector + `PATH` lookup.
- `execvpe(file, argv[], envp[])` — Vector + `PATH` + custom environment (GNU-only).

Function	Use When	PATH Lookup
`execl`	Static args, full path	No
`execlp`	Static args, uses `PATH`	Yes
`execle`	Static args, custom env	No
`execv`	Dynamic args, full path	No
`execvp`	Dynamic args, uses `PATH`	Yes
`execvpe`	Dynamic args, `PATH`, env	Yes (GNU-only)

- `pthread_create()` spawns a thread that shares the same address space.
- `pthread_join()` waits for a thread to finish.
- Semaphores (`sem_t`) are integer counters used to control access to shared resources.
- `sem_init(sem, [0(thread)|1(process)], value)` — Initialize semaphore to given value.
- `sem_wait(sem)` — Decrement; blocks if value is 0.
- `sem_post(sem)` — Increment; unblocks one waiter if any.
- `sem_destroy(sem)` — Cleans up; does not free memory.
- Used for mutual exclusion (binary semaphores) or limiting access (counting semaphores).

Dining Philosophers: Limit-Seat Strategy

- Using a semaphore initialized to $N-1$ prevents deadlock in the dining philosopher problem.
- Limits the number of philosophers who can attempt to pick up chopsticks to ensure progress.
- Prevents circular wait, breaking one of Coffman's deadlock conditions.
- Starvation is still possible due to unfair scheduling.

```
typedef struct {
    int _status[N];
    sem_t mutex;
    sem_t sem[N];
} SharedMem;
void takeChpStk(SharedMem* shm, int i) {
    sem_wait(&shm->mutex);
    shm->_status[i] = HUNGRY;
    safeToEat(shm, i);
    sem_post(&shm->mutex);
    sem_wait(&shm->sem[i]);
}
void safeToEat(SharedMem* shm, int i) {
    if ((shm->_status[i] == HUNGRY) &&
        (shm->_status[LEFT] != EATING) &&
        (shm->_status[RIGHT] != EATING)) {
            shm->_status[i] = EATING;
            sem_post(&shm->sem[i]);
        }
}
void putChpStk(SharedMem* shm, int i) {
    sem_wait(&shm->mutex);
    shm->_status[i] = THINKING;
    safeToEat(shm, LEFT);
    safeToEat(shm, RIGHT);
    sem_post(&shm->mutex);
}
```

- **Turnaround t**: Total time from job arrival to completion.
- **Response t**: Time from job arrival to first CPU execution.
- **Waiting time**: Time a job spends in the ready queue.
- **Throughput**: Number of jobs completed per unit time.

Algorithm	Preemptive	Fairness	Response Time
FCFS	No	Arrival-order fair	Poor
SJF	No	No	Excellent for short jobs
SRT	Yes	No	Best for short jobs
RR	Yes	Yes	Good
Lottery	Optional	Probabilistic	Fair on average
MLFQ	Yes	Adaptive	Excellent

Algorithm	Turnaround Time	Starvation Risk
FCFS	High (convoy effect)	Low
SJF	Optimal (theoretical)	High
SRT	Optimal	High
RR	Medium (depends on quantum)	Low
Lottery	Fair on average	Low
MLFQ	Adaptive	Moderate (if not tuned)

Table storage:

- UPP: user process pages
- SWAP: Non-memory resident iser process page
- Process page table: in PCB table in OS mem region in RAM
- Open file table: in OS mem region in RAM
- File descriptor table: in PCB
- Dynamically allocated mem in a prgram: UPP or SWAP
- file decriptor returned from an open(...) syscall: UPP or SWAP
- compiled binary files: not part of the virtual mem

Contiguous mem:

- **Tracking free space:**
  - Bitmap: 1 bit per block, where 0 = free, 1 = allocated.
  - Linked List: Each free block links to the next.
  - Buddy System: Memory is split into power-of-2 blocks; recursive splitting/merging.
- **Fragmentation:**
  - Internal: Block larger than needed.
  - External: Gaps between allocated blocks.

Paging

- Fixed-size units: Logical pages and physical frames.
- Page Table: Maps pages to frames.
- TLB: Hardware cache for recent page table entries.

Segmentation

- Logical memory divided into named segments (code, stack, heap).
- Each has a base and limit.
- Logical Address = <Segment ID, Offset>.

Virtual mem

- Logical memory can exceed physical memory.
- Disk serves as backing store.

Demand Paging

- Pages are only loaded on access. No memory resident page
- (+) Reduces startup time and memory usage.
- (-) more page fault at start; page fault can cascade on other processes (e.g. thrashing)

Page Access

```
Check page table:
    if memory resident: acess physical mem; done;
    else: [page fault] -> trap to OS
        locate page in secondary storage;
        load into physical mem;
        update page table;
        goto Check page table;
```

Single-Level

- Flat array of entries.
- Wasteful for sparse address spaces.

Multilevel

- Use a page directory pointing to page tables.
- Only allocate when needed.
- page dir base reg $\rightarrow$`<page_dir#, page#, ofst>`
- overhead = sizeof(page_dir) + $\sum$sizeof(small_pagetable)

Inverted Page Table

- One entry per frame: `<pid, page#>` $\rightarrow$ frame.
- Compact but slow due to full-table lookup.

Feature	Direct Paging
Page Table Size	Grows linearly with virtual space
Lookup Cost	Fast (1 access + TLB)
Translation Overhead	Simple calculation
Space Efficiency	Poor with sparse address spaces
Entry Granularity	One entry per virtual page
Process Isolation	Each process has separate page table

Multilevel Paging	Inverted Page Table
Compact, only allocates needed tables	Fixed size (per physical frame)
Slower (multi-level lookup)	Slow unless hashed (may need full scan)
Multiple memory accesses	Needs reverse mapping
Good for sparse address spaces	Excellent for large sparse spaces
One entry per virtual page	One entry per physical frame
Each process has separate page table	Global table with PID tag

- **Temporal Locality:** Recently used memory will be used again.
- **Spatial Locality:** Nearby memory addresses are likely to be used soon.

Page Replacement Algorithms

- OPT: Replace page with furthest next use (ideal).
- FIFO: Oldest page out.
- LRU: Least recently used page.
- Clock: Approximate LRU using reference bits.

$$T_{access} = (1 - p) \cdot T_{mem} + p \cdot T_{page\_fault}$$

Local Replacement

- Only evict pages from the same process.
- Predictable and isolated.
- if not enf allocated, hinders process progress

Global Replacement

- Victim page can belong to any process.
- More flexible, allows self-adjustment, but less stable.
- bad behaved process can affect others

Thrashing

- Excessive page faults reduce CPU utilization.
- Can lead to cascading faults in global replacement.
- **Working Set Model:** Allocate enough frames for `W(t, Δ)`.

- A file is the smallest amount of information that can be written to secondary memory. It is a named collection of data, used for organizing secondary memory
- A file type is a description of the information contained in the file. A file extension is a part of the file name that follows a dot and identifies the file type
- What does it mean to open and close a file? Operating systems keep a table of currently open files. The open operation enters the file into this table and places the file pointer at the beginning of the file. The close operation removes the file from the table of open files.
- Truncating a file means that all the information on the file is erased but the administrative entries remain in the file tables. Occasionally, the truncate operation removes the information from the file pointer to the end.

Aspect	**Memory Management**
Underlying Storage	RAM
Access Speed	Constant
Unit of Addressing	Physical memory address
Usage	Address space for process **Implicit** when process runs
Organization	**Paging/Segmentation**: determined by HW & OS

Aspect	**File System Management**
Underlying Storage	Disk
Access Speed	Variable disk I/O time
Unit of Addressing	Disk sector
Usage	Non-volatile data **Explicit** access
Organization	**Many FS types**: ext* (Linux), FAT* (Windows), HFS* (Mac)

- **Name**: A human-readable reference to the file.
- **Identifier**: A unique ID for the file used internally by the file system.
- **Type**: Indicates the type of file (e.g., executable, text file, object file, directory, etc.).
- **Size**: Current size of the file (in bytes, words, or blocks).
- **Protection**: Access permissions, which may include reading, writing, and execution rights.
- **Time, date, and owner information**: Includes creation time, last modification time, owner ID, etc.
- **Table of content**: Metadata that enables the file system to determine how to access the file.

- A process uses the `open()` system call to access a file:
  - Example: `int fd = open("data.txt", O_RDONLY);`
  - Returns a file descriptor (`fd`), an integer index into the process's file descriptor table.
- Internally, the OS performs:
  - Path resolution and access permission check.
  - Loads file metadata (e.g. inode) into memory.
  - Creates an entry in the **System-wide Open File Table**, including:
    - File offset (initially 0)
    - Pointer to file metadata (inode)
    - File mode (read, write, etc.)
  - Updates the process's **Per-Process File Descriptor Table**:
    - `fd` points to the corresponding system-wide table entry.
- Shared file descriptors:
  - Two `fd`s pointing to the same system-wide entry share offset and metadata.
  - Created using `dup()`, `dup2()`, or inherited from `fork()`.
- Accessing file info:
  - Use `fcntl(fd, F_GETFL)` to query file status flags.
  - Use `lseek(fd, 0, SEEK_CUR)` to query current offset.

Feature	**Contiguous**	**Linked List**	**FAT**	**Inode-based (e.g., ext)**
Access time (random)	Fast	Slow	Moderate	Fast
Access time (sequential)	Fast	Fast	Fast	Fast
Disk fragmentation	High	None	None	Low
Supports random access	Yes	No	Yes (with FAT table)	Yes
Space efficiency	Poor	Good	Good	Very good
Pointer overhead	None	High	High (FAT in memory)	Low (indirect blocks)
File size flexibility	Poor	Good	Good	Excellent
Crash recovery	Poor	Poor	Moderate	Good (journaling)

Feature	**FAT**	**EXT (inode)**	**NTFS (MFT)**
Allocation method	FAT table (linked list)	Inode + indirect blocks	Extents + B-tree indexing
Metadata location	Centralized table	Distributed inodes	MFT entries
Scalability	Poor for large disks	Very scalable	Highly scalable
Crash tolerance	Low (no journaling)	High (journaling via ext3/4)	High (journaling)
Maximum file size	Limited	Large (ext4: 16 TiB)	Very large
Directory management	Linear list	Hash tree (ext4)	B-tree

Feature	**Hard Link**	**Symbolic Link**
Points to	Inode (actual file)	File path (string)
Requires own inode	No	Yes
Spans file systems	No	Yes
Can link to directory	No	Yes
Broken if target deleted	No	Yes (becomes dangling)
Deletes actual file?	Only if last hard link removed	No
ls -l output	Normal file	l with → path

- `open(const char *pathname, int flags[,mode])`:
  - Opens a file and returns a file descriptor (int).
  - `flags` specify access mode: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, etc.
  - `mode` is required if `O_CREAT` is used, to set permission bits.
- `read(int fd, void *buf, size_t count)`:
  - Reads up to `count` bytes from file descriptor `fd` into buffer `buf`.
  - Returns the number of bytes read, or 0 on EOF.
  - Advances the file offset by the number of bytes read.
- `write(int fd, const void *buf, size_t count)`:
  - Writes up to `count` bytes from buffer `buf` to file descriptor `fd`.
  - Returns the number of bytes written.
  - Advances the file offset by the number of bytes written.
- `lseek(int fd, off_t offset, int whence)`:
  - Moves the file offset for `fd`.
  - `whence` can be `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`.
  - Returns the new offset, or `-1` on error.
- `close(int fd)`:
  - Closes the file descriptor `fd`.
  - Releases the file table entry and associated kernel resources.
  - Returns 0 on success, `-1` on error.