

# Scheduling Algorithms

## FCFS

- Process added first in ready queue should be scheduled first.
- Non-preemptive scheduling
- Scheduler is invoked when process is terminated, blocked or gives up CPU is ready for execution.
- Convoy Effect: Larger processes slow down execution of other processes.

## SJF

- Process with lowest burst time is scheduled first.
- Non-preemptive scheduling
- Minimum waiting time

## SRTF - Shortest Remaining Time First

- Similar to SJF - but Preemptive scheduling
- Minimum waiting time

## Priority

- Each process is associated with some priority level. Usually lower the number, higher is the priority.
- Preemptive scheduling or Non Preemptive scheduling
- Starvation
  - Problem may arise in priority scheduling.
  - Process not getting CPU time due to other high priority processes.
  - Process is in ready state (ready queue).
  - May be handled with aging -- dynamically increasing priority of the process.

## Round-Robin

- Preemptive scheduling
- Process is assigned a time quantum/slice.
- Once time slice is completed/expired, then process is forcibly preempted and other process is scheduled.
- Min response time.

## Fair-share

- CPU time is divided into epoch times.
- Each ready process gets some time share in each epoch time.
- Process is assigned a time share in proportion with its priority.
- In Linux, processes with time-sharing (TS) class have nice value. Range of nice value is -20 (highest priority) to +19 (lowest priority).

## IPC overview

- A process cannot access the memory of another process directly. OS provides IPC mechanisms so that processes can communicate with each other.
- IPC models
  - Shared memory model
    - Processes write/read from the memory region accessible to both the processes.
    - OS only provides access to the shared memory region.
  - Message passing model
    - Process send message to the OS and the other process receives message from the OS.
    - This is slower compared to shared memory model.
- Unix/Linux IPC mechanisms
  - Signals
  - Shared memory
  - Message queue
  - Pipe
  - Socket

## Message Queue

- Used to transfer packets of data from one process to another.
- It is bi-directional IPC mechanism.
- Internally OS maintains list of messages called as "message queue" or "mailbox".
- The info about msg que is stored in a object. It contains unique KEY, permissions, message list, number of messages in list, processes waiting for a message to receive (waiting queue).

## Message Queue Syscalls

- msgget()
  - Create message queue object.
  - mqid = msgget(mq\_key, flags);
    - arg1: unique key
    - arg2: IPC\_CREAT | 0600 to create new message queue.
    - returns message queue id on success
- msgctl()
  - Get info about message queue or destroy message queue
  - msgctl(mqid, IPC\_RMID, NULL) -- to destroy message queue
    - arg1: message queue id
    - arg2: command = IPC\_RMID to destroy message queue
    - arg3: NULL (not required while destroying message queue)
- msgsnd() - send message into que
  - Send message in the message queue.
  - ret = msgsnd(mqid, msg\_addr, msg\_size, flags)
    - arg1: message queue id
    - arg2: base address of message object
    - arg3: message body/payload size
    - arg4: flags (=0 for default behaviour)
    - returns 0 on success.
- msgrcv() - receive message from que

- Receive message from the message queue of given type.
- `ret = msgrcv(mqid, msg_addr, msg_size, msg_type, flags)`
  - `arg1`: message queue id
  - `arg2`: base address of message object to collect message (out param)
  - `arg3`: message body/payload size
  - `arg4`: type of message to be received
  - `arg5`: flags (=0 for default behaviour)
  - returns number of bytes (body) received on success.

## Signals

- OS have a set of predefined signals, which can be displayed using command
  - `terminal> kill -l`
- A process can send signal to another process or OS can send signal to any process.
- Information about signals.
  - `terminal> man 7 signal`

## Send signal

- `kill` command is used to send signal to another process, which internally use `kill()` syscall.
  - `terminal> kill -SIG pid`
- `pkill` command is used to send signal to multiple processes/instances of the same program.
  - `terminal> pkill -SIG programname`

## Imporant Signals

1. **SIGINT (2)**: When **CTRL+C** is pressed, **INT** signal is sent to the foreground process.
2. **SIGTERM (15)**: During system shutdown, OS send this signal to all processes. Process can handle this signal to close resources and get terminated.
3. **SIGKILL (9)**: During system shutdown, OS send this signal to all processes to forcefully kill them. Process cannot handle this signal.
4. **SIGSTOP (19)**: Pressing **CTRL+S**, generate this signal which suspend the foreground process. Process cannot handle this signal.
5. **SIGCONT (18)**: Pressing **CTRL+Q**, generate this signal which resume suspended the process.
6. **SIGSEGV (11)**: If process access invalid memory address (dangling pointer), OS send this signal to process causing process to get terminated. It prints error message "Segmentation Fault".
7. **SIGCHLD (17)**: When child process is terminated, this signal is sent to the parent process. The parent process may handle this to get the exit code of the child (`wait()` syscall).
8. **SIGHUP (1)**: When a terminal is closed, all processes running in that terminal and terminated due to Hang up signal.

## Signals related syscalls

- `kill()` syscall
  - `kill()` send signal to another proces.
  - `ret = kill(pid, signum);`
    - `arg1`: pid of the process to whom signal is to be sent.
    - `arg2`: signal number -- defined in `signal.h`

- returns: 0 on success and -1 on failure.
- signal() syscall
  - signal() is used to install signal handler in current process (signal handler table).
  - When signal is received, OS calls registered signal handler.
  - old\_handler = signal(signum, new\_handler);
    - arg1 (int): signal number of signal to be handled (except SIGKILL and SIGSTOP).
    - arg2 (fn ptr): address of signal handler function.
      - typedef void (\*sighandler\_t)(int);
    - returns: address of old signal handler (in the table).

## File IO syscalls

- open()
- read()
- write()
- close()
- lseek()

### open() syscall

- fd = open("file-path", flags, mode);
  - arg1: path of file to be opened
  - arg2: flags - how you want to open the file
    - O\_RDONLY, O\_WRONLY, O\_RDWR -- read-write flags
    - O\_TRUNC -- delete the contents of file while opening
    - O\_APPEND -- write at the end of file
    - O\_CREAT -- create a new file (if not present) -- must give arg3
  - arg3: mode - permissions for new file - octal number
  - returns file descriptor on success, and -1 on failure.
    - file descriptor is int that uniquely identifies the file in that process.
    - fd is used in other file io syscalls e.g. close(), read(), write(), lseek().

### close() syscall

- int close(fd)
  - arg1: fd returned by open().
  - Returns 0 on success and -1 on error
- Decrement ref count in open file table entry (struct file).
- If ref count drops to zero, OFT entry is deleted (from OFT).

### read() syscall

- count = read(fd, buf, length);
  - arg1: file descriptor
  - arg2: buffer in which you want to read
  - arg3: length of buffer
  - Returns: Number of characters read

## write() syscall

- `ssize_t write(int fd, const void *buf, size_t count);`
  - arg1: file descriptor
  - arg2: buffer from which you want to write
  - arg3: length of data
  - Returns: Number of characters written

## lseek() syscall

- Change the file position in open file table entry (struct file --> `f_pos`).
- And returns new file position (from the beginning of the file).
- `off_t lseek(int fd, off_t offset, int whence);`
- Examples:
  - `lseek(fd, 0, SEEK_SET);`
    - `filp->f_pos = 0;`
  - `lseek(fd, offset, SEEK_SET);`
    - `filp->f_pos = offset;`
  - `lseek(fd, 0, SEEK_END);`
    - `filp->f_pos = size; // file size (from the inode)`
  - `lseek(fd, offset, SEEK_END);`
    - `filp->f_pos = size + offset; // note that offset will be negative`
  - `lseek(fd, offset, SEEK_CUR);`
    - `filp->f_pos = filp->f_pos + offset;`

## Pipe

- Pipe is used to communicate between two processes.
- It is stream based uni-directional communication.
- Pipe is internally implemented as a kernel buffer, in which data can be written/read.
- If pipe (buffer) is empty, reading process will be blocked.
- If pipe (buffer) is full, writing process will be blocked.
- If writer process is terminated, reader process will read the data from pipe buffer and then will get EOF.
- If reading process is terminated, writing process will receive SIGPIPE signal.
- There are two types of pipe:
  - Unnamed Pipe
  - Named Pipe

## pipe() syscall

- To create unnamed pipe.
- `ret = pipe(fd); // int fd[2];`
  - arg1: array of two ints to collect fd (out param).
  - returns 0 on success.
  - `fd[]` gets two ends of pipe.
    - `fd[0]` -- read end (file descriptor) of pipe
    - `fd[1]` -- write end (file descriptor) of the file