04. Threads

Created	@February 17, 2022 7:07 PM
Tags	Lee Edith.D

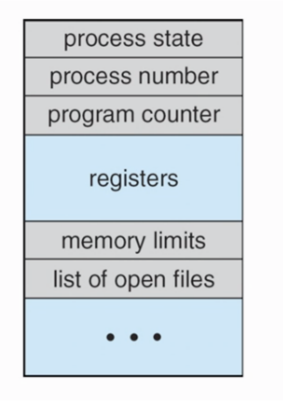
▼ review：

Process

> for OS，how to manage so many processes?

memory hierarchy for a process

- PCB



PCB

1. **seems** to have whole memory!
2. some parts can't visit

   OS code, PCB description, Kernel Stack(used when jump instruction execute)

- linux task_struct

process scheduling / process management API

- exit

  delete nearly all space of a process, but not included current process's PCB

- fork

  set up a new PCB, copy PARTS of the father process like files, mm... not include process number, it should be a new one.

- exec

  rebuild the process by other programs, cover code, data, stack... in original PCB

- wait

  hang up father process until son process finished, kill son process

signal handlers principles

> signal is all about ‘software interrupt’, OS captures the signal and transports this to a specific process

control code is in PCB. There’s a caption about signal handlers. **signal *shand*

ex. KILL to send signal

1. command

   ```
   kill -9 pid_number
   ```

2. system schedule

   ```
   int(int pid, int sig)
   ```

how to receive signal for OS

when OS is ready for transferring the control rights to process p, calculating pnb=pending & ~blocked, this means to ensure whether if there’s a signal to interrupt and execute.

▼ IPC (Interprocess Communication)

> everything is a file, *files on disk, external devices, networking(sockets), pipes...*

file abstraction

- file: collection of data in a file system
- directory: “folder” containing files

Relationships between files and process

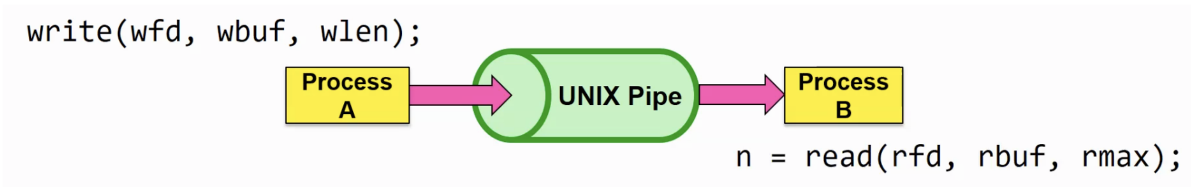
- CWD (current working directory) is the basic property of a directory

Buffer

- why? reduce operations from OS to Usr
- `fopen` will lead to a new space with **buffer**, **file description signal,** data will firstly get stored into buffer before get refreshed in OS

Pipe

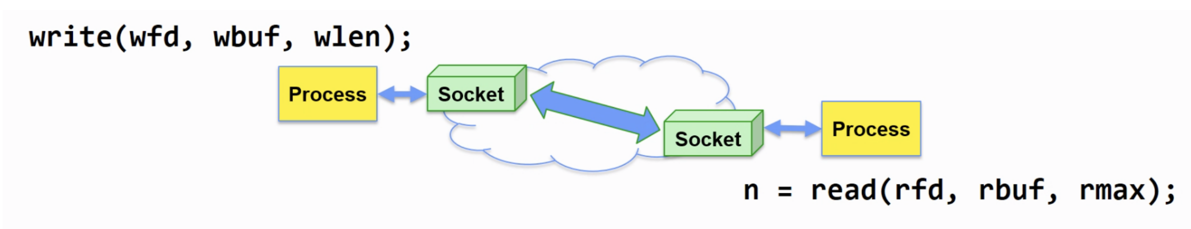
shared memory




```
int pipe(int fileds[2])
```

- queue
- `pipe[0]`, `pipe[1]`

Socket

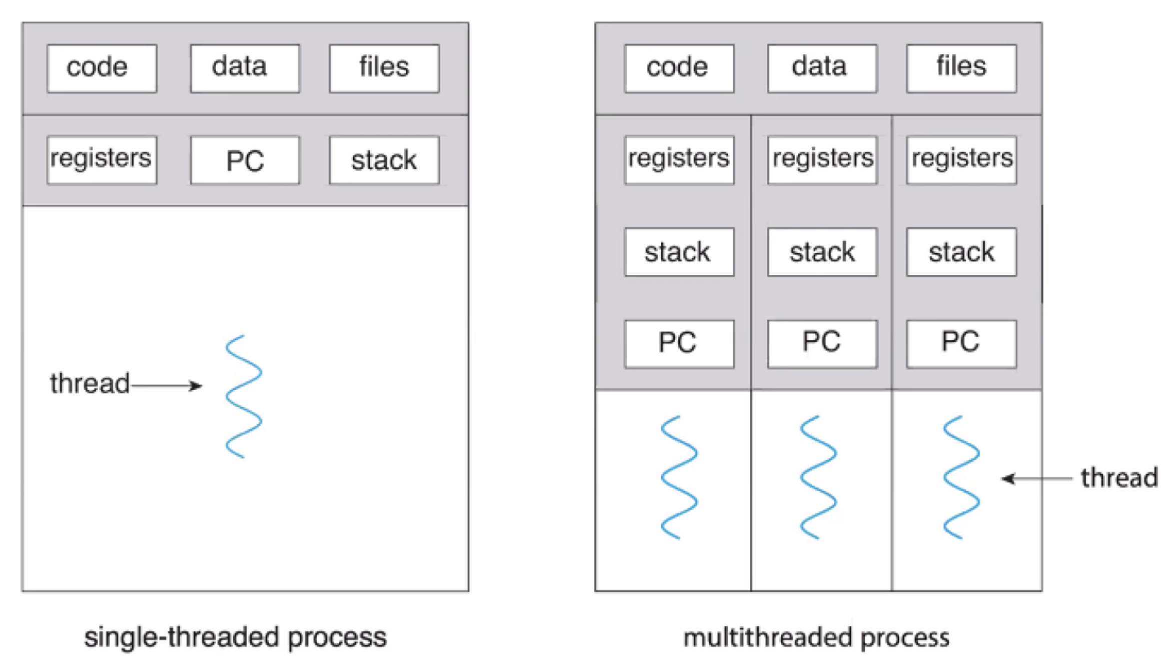


- queue

Backgrounds

- thread is the basic **sequential** unit in CPUs
- in a process, threads
  - share

        - code
        - data
        - resources (files...)
    - easy to communicate
    - context switch is easy



- multi-threads leads to multi-cores processor
- includes different:
    - thread ID
    - PC
    - **stack**
    - registers
- **can concurrently execute programs by different threads** tasks in a program can be divided many threads, improve 资源利用率
- **can hide I/O delay**

Module

User Level Threads

- above kernel, no need to have support from kernel
- kernel can only see process list
    - do not participate thread allocating
    - thread allocating is implemented by user program
- cons: if thread has blocked, the process will be blocked

Kernel Level Threads

- supported and managed by OS kernel
- system calls
- cons: cost

Many-to-One

- Many usr-threads are mapping to a single kernel thread
- cons:
    - if one usr-thread block → corresponding kernel thread blocks → the other usr-threads block
    - not for many-core system
- **OS schedule kernel thread**
- eg: Former Java

One-to-One

- Each usr-thread is mapping to a single kernel-thread
- cons:

- cost

- eg: Linux

Many-to-Many

- many usr-threads - many kernel-thread

Thread Library

Pthread

- usr-thread
- kernel-thread
- provide API
- funcs:
  - `pthread_create()`
  - `pthread_join()`
- pthread & system_call:

Thread in Linux

- LWP (Lightweight Process)
- `fork()` and `pthread_create()` both call for system call `clone()`
  - but `pthread_create()` calls for `mmap()` for different **stacks**