Operating Systems: Processes and Threads

keleher, shankar

February 6, 2024
1. Process State
2. Process Creation
3. Process Termination
4. User-Threads Management
5. Booting the OS
6. Inter-Process Communication: Pipes
7. Inter-Process Communication: Signals
8. Inter-Process Communication: Internet Sockets
9. Schedulers
- Process: executing instance of a program
  - Threads: active agents of a process
- Address space
  - text segment: code
  - data segment: global and static
  - stack segment, one per thread
- Resources: open files, sockets, pipes
- Code: non-privileged instructions
  - including syscalls to access OS services
- All threads execute concurrently (scheduling undefined)
Figure 4.1: Loading: From Program To Process
In the OS Kernel

- Data structures: state of processes
- Process: address space, resources, threads ("kernel threads")
  - kernel thread: user-stack, kernel-stack, processor state
  - user thread: user-stack, only per-process kernel-stack, not visible to kernel
- mapping of content to hardware location (e.g., memory, disk)
  - memory vs disk (swapped out)
- thread status: running, ready, waiting, mode
- kernel process: kernel-stack, processor state, no user-level visibility

- Schedulers, queues:
  - short-term: ready → running
  - io device: waiting → io service → ready
  - medium-term: ready/waiting ↔ swapped-out
  - long-term: start → ready
- efficiency and responsiveness
Single-Threaded Process

- PCB (process control block): one per process
  - holds enough state to resume the process
  - process id (pid)
  - processor state: gpr, ip, ps, sp, ...
  - address-space: text, data, user-stack, kernel-stack
    - mapping to memory/disk
  - io state: open files.Sockets, current positions, access, ...
  - accounting info: processor time, memory limits, ...
  - ...

- Status
  - running: executing on a processor
  - ready (aka runnable): waiting for a processor
  - waiting: for a non-processor resource (eg, memory, io, ...)
  - swapped-out: holds no memory
Multi-Threaded Process

- PCB (process control block): one per process
  - address-space: text, data
  - io state
  - accounting info
- TCBs (thread control block): one per thread
  - processor state
  - user-stack, kernel-stack
  - status: running, ready, waiting, ...
- ...  
- Process swapped-out → all threads swapped out
- Kernel threads operate in two contexts:
  - user-mode: executing user code, using user-stack
  - kernel-mode: executing kernel code, using kernel-stack
- Process that runs only in the kernel
  - asynchronous services: io, reaper, ...
  - always in kernel-mode
  - TCB (thread control block): one per kernel thread
    - holds enough state to resume the thread
    - processor state: gpr, ip, ps, sp, ...
    - kernel-stack // no user-stack
- status: running, ready, waiting
User threads

- Threads implemented entirely in user process
- not visible or schedulable by kernel
  - process might have multiple user threads
  - but kernel only sees one

- User code maintains
  - TCBs
  - signal handlers (for timer/io/etc interrupts)
  - dispatcher, scheduler

- OS provides low-level functions via which user process can
  - get processor state
  - dispatch processor state
  - to/from environment variables

- User-level vs kernel-level
  - Pro: application-specific scheduling
  - Con: cannot exploit additional processors
Different types of threads:
- *kernel threads* - can be seen and scheduled by the kernel, have both user and kernel stacks
- *user threads* - not visible to kernel, only user stacks

Also *kernel processes* - threads that execute only in the OS kernel
- term not used as much as the above
- not user visible
- only kernel stack
Process queues

- Kernel keeps PCBs/TCBs in queues
  - new queue: processes to be started
  - run queue
  - ready (aka runnable) queue
  - io queue(s)
  - swapped-out queue
  - terminated queue: processes to be cleaned up

- Transitions between queues
Outline

1. Process State
2. Process Creation
3. Process Termination
4. User-Threads Management
5. Booting the OS
6. Inter-Process Communication: Pipes
7. Inter-Process Communication: Signals
8. Inter-Process Communication: Internet Sockets
9. Schedulers
Approach 1: Create Process from Scratch

CreateProcess(*path, context*):
- read file from file system’s *path*
- acquire memory segments
- unpack file into its segments
- create PCB
- update PCB with *context*
- add PCB to ready queue

// GeekOS Spawn()
// executable file
// code, data, stack(s), ...
// pid, ...
// user, directory, ...

Drawback: *context* has a lot of parameters to set

Your version of GeekOS only has this type of process creation
Approach 2: Fork-Exec

Fork(): creates a copy of the caller process
// returns 0 to child, and child’s pid to parent
- create a duplicate PCB
  - except for pid, accounting, pending signals, timers, outstanding io operations, memory locks, ...
  - only one thread in new process (the one that called fork)
- allocate memory and copy parent’s segments
  - minimize overhead: copy-on-write; memory-map hardware
  - add PCB to the ready queue

Exec(path, ...): replaces all segments of executing process
- exec[elpv] variants: different ways to pass args, ...
- open files are inherited
- not inherited: pending signals, signal handlers, timers, memory locks, ...
- environment variables are inherited except with exec[lv]e

Project 1
1. Process State
2. Process Creation
3. Process Termination
4. User-Threads Management
5. Booting the OS
6. Inter-Process Communication: Pipes
7. Inter-Process Communication: Signals
8. Inter-Process Communication: Internet Sockets
9. Schedulers
Process A becomes a zombie when

- A executes relevant OS code (intentionally or o/w)
  - exit syscall
  - illegal op
  - exceeds resource limits
  - ...
- A gets kill signal from a (ancestor) process

A is moved to terminated queue

What happens to A’s child process?

- becomes a root process’s child (orphan)  // Unix
- is terminated // VMS
Zombie process A is eventually *reaped*

- its memory is freed
- its parent is signalled (SIGCHILD)
- it waits for parent to do `wait()` syscall
  - parent gets exit status, accounting info, ...
Outline

1. Process State
2. Process Creation
3. Process Termination
4. User-Threads Management
5. Booting the OS
6. Inter-Process Communication: Pipes
7. Inter-Process Communication: Signals
8. Inter-Process Communication: Internet Sockets
9. Schedulers
POSIX threads

- `thread_create(thrd, func, arg)`
  - create a new user thread executing `func(arg)`
  - return pointer to thread info in `thrd`

- `thread_yield()`:
  - calling thread goes from running to ready
  - scheduler will resume it later

- `thread_join(thrd)`:
  - wait for thread `thrd` to finish
  - return its exit code

- `thread_exit(rval)`:
  - terminate caller thread, set caller’s exit code to `rval`
  - if a thread is waiting to join, resume that thread

- POSIX threads is an API (not implementation) definition
  - can be implemented either as user threads, or kernel threads
Outline

1. Process State
2. Process Creation
3. Process Termination
4. User-Threads Management
5. Booting the OS
6. Inter-Process Communication: Pipes
7. Inter-Process Communication: Signals
8. Inter-Process Communication: Internet Sockets
9. Schedulers
OS initialization

- **Power-up:**
  - BIOS: disk boot sector → RAM reset address
  - processor starts executing contents

- **Boot-sector code:**
  - load kernel code from disk sectors to RAM, start executing

- **Kernel initialization:**
  - identify hardware: memory size, io adaptors, ...
  - partition memory: kernel, free, ...
  - initialize structures: vm/mmap/io tables, pcb queues, ...
  - start daemons: OS processes that run in the background
    - idle
    - io-servers
    - login/shell process bound to console
  - mount filesystem(s) in io device(s)
1. Process State
2. Process Creation
3. Process Termination
4. User-Threads Management
5. Booting the OS
6. Inter-Process Communication: Pipes
7. Inter-Process Communication: Signals
8. Inter-Process Communication: Internet Sockets
9. Schedulers
Kernel file data structures

- **Inode table**: has a copy of the inode of every open vertex (file or directory)
  - may differ from the inode in the disk
- **Open-file table**: has an entry for every open call not yet succeeded by a close call (across all processes)

  Each entry holds:
  - current file position, reference count (how many file descriptors point to the entry), inode pointer, etc.
  - Entry is removed when the reference count is 0

- For each process: a file descriptor table, mapping integers to open-file table entries

The open-files-table is system wide, and only has refs>1 after fork() or dup().
Opening the same file twice

```c
fd1 = open("file.txt", O_RDONLY);
fd2 = open("file.txt", O_RDONLY);
read(fd2, buffer, 1024);
```
After a `fork()`

```c
fd1 = open("file.txt", O_RDONLY);
fd2 = open("file.txt", O_RDONLY);
read(fd2, buffer, 1024);
fork();
```

- **Parent**
  - open file table
    - position: 0
    - ref. count: 2
    - inode
    - permissions: 0666
    - size: 50238
    - type: regular file
  - ...
  - 3
  - 4

- **Child**
  - open file table
    - position: 1024
    - ref. count: 2
    - inode
    - ...
  - ...
  - 3
  - 4

© 2016 L. Herman & A. U. Shankar
Opening a pipe

```c
int pfd[2];
pipe(pfd);
```
After a `fork()`

```c
int pfd[2];
pipe(pfd);
fork();
```

**Open file table**

- Position: n/a
- Reference count: 2
- Inode

**Inode table entry**

- Permissions: 0666
- Size: 0
- Type: Pipe

---

Example `pipe-example.c`
Example: data transfer on pipe from parent to child

- Process, say \(A\), creates pipe
- \(A\) forks, creating child process, say \(B\)
- \(A\) closes its read-end of pipe, writes to pipe
- \(B\) closes its write-end of pipe, reads from pipe
- byte stream: in-chunks need not equal out-chunks
- \(A\) blocks if buffer is full and \(B\) has not closed read-end
- \(B\) blocks if buffer is empty and \(A\) has not closed write-end

- read when no data and no writers (write-end has zero ref count):
  - read returns 0
- write when no readers (read-end has zero ref count):
  - writer process receives SIGPIPE signal
  - write returns EPIPE