Operating Systems: Processes and Threads

Shankar

February 10, 2022

Outline

User Perspective

- 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 and sockets
- Code: non-privileged instructions
  - including syscalls to access OS services
- All threads execute concurrently

OS Kernel

- Data structure: state of processes, user threads, kernel threads
- Process: address space, resources, user threads
  - user thread: user-stack, kernel-stack, processor state
  - mapping of content to hardware location (eg, memory, disk)
    - memory vs disk (swapped out)
  - user thread status: running, ready, waiting, mode
- Kernel thread: kernel-stack, processor state

- Schedulers:
  - short-term: ready → running
  - io device: waiting → io service → ready
  - medium-term: ready/waiting ↔ swapped-out
  - long-term: start → ready
  - efficency and responsiveness

Outline

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 // user thread
    - processor state
    - user-stack, kernel-stack
    - status: running, ready, waiting, ...
  - ...
- Process swapped-out → all threads swapped out
- User thread:
  - user-mode: executing user code, using user-stack
  - kernel-mode: executing kernel code, using kernel-stack

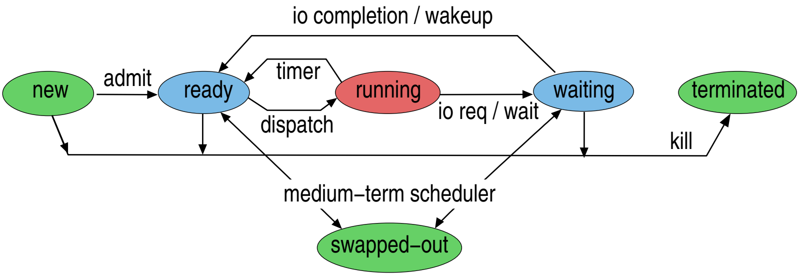
Kernel threads

- Threads belonging to 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

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



User-level Threads

- Threads implemented entirely in user process
- Kernel is not aware of them
  - kernel sees only one user thread

- 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

Outline

Approach 1: Create Process from Scratch

- CreateProcess(*path*, *context*): // GeekOS Spawn
  - read file from file system's *path* // executable file
  - acquire memory segments // code, data, stack(s), ...
  - unpack file into its segments
  - create PCB // pid, ...
  - update PCB with *context* // user, directory, ...
  - add PCB to ready queue
- Drawback: *context* has a lot of parameters to set

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 (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

Outline

Zombie

- 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 (if any)
  - becomes a root process's child (orphan) // Unix
  - is terminated // VMS

Reap

- Process *A* in the termination queue 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. Overview
2. Process State
3. Process Creation
4. Process Termination
5. User-Threads Management
6. Booting the OS
7. Inter-Process Communication: Pipes
8. Inter-Process Communication: Signals
9. Inter-Process Communication: Internet Sockets
10. 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

Outline

1. Overview
2. Process State
3. Process Creation
4. Process Termination
5. User-Threads Management
6. Booting the OS
7. Inter-Process Communication: Pipes
8. Inter-Process Communication: Signals
9. Inter-Process Communication: Internet Sockets
10. 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)

Outline

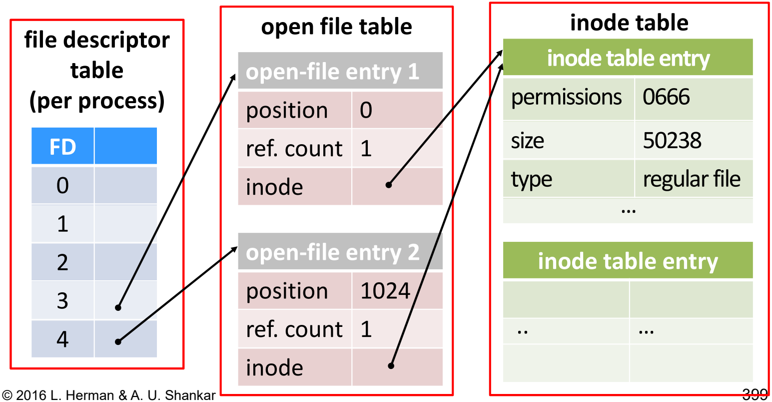
Kernel file data structures

- 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

Opening the same file twice

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



file descriptor table (per process)

FD	
0	
1	
2	
3	●
4	●

open file table

open-file entry 1	
position	0
ref. count	1
inode	●

open-file entry 2	
position	1024
ref. count	1
inode	●

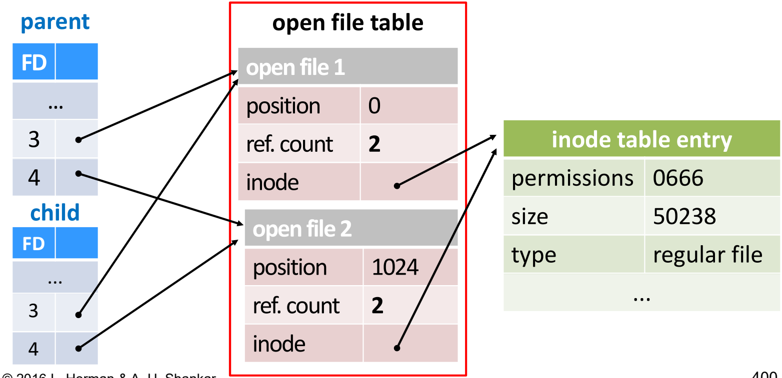
inode table

inode table entry	
permissions	0666
size	50238
type	regular file
...	

inode table entry	
..	...

After a `fork()`

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



parent

FD	
…	
3	
4	

child

FD	
…	
3	
4	

open file table

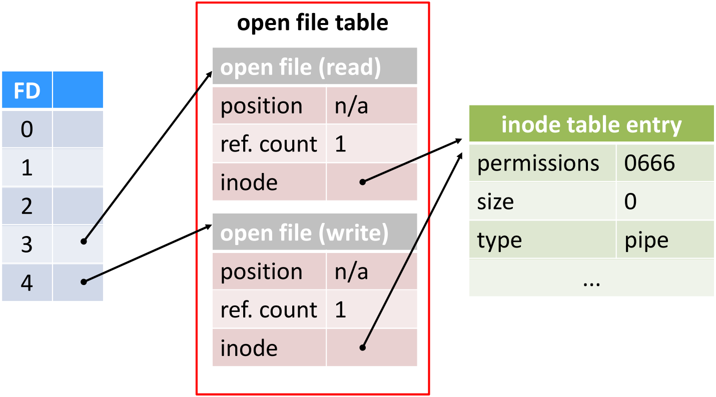
open file 1	
position	0
ref. count	**2**
inode	

open file 2	
position	1024
ref. count	**2**
inode	

inode table entry	
permissions	0666
size	50238
type	regular file
…	

Opening a pipe

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



FD	
0	
1	
2	
3	
4	

open file table

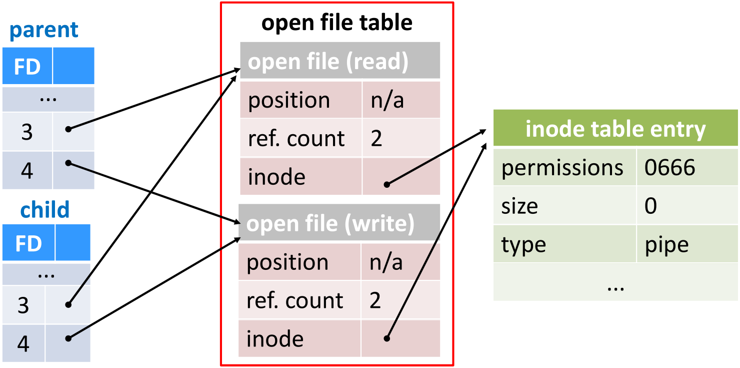
open file (read)	
position	n/a
ref. count	1
inode	

open file (write)	
position	n/a
ref. count	1
inode	

inode table entry	
permissions	0666
size	0
type	pipe
...	

After a `fork()`

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



parent

FD	
…	
3	•
4	•

child

FD	
…	
3	•
4	•

open file table

open file (read)	
position	n/a
ref. count	2
inode	•

open file (write)	
position	n/a
ref. 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

Outline

Signals: user perspective

- Process-level interrupt with a small integer argument *n* (0..255)
  - SIGKILL, SIGCHILD, SIGSTOP, SIGSEGV, SIGILL, SIGPIPE, ...
- Who can send a signal to a process *P*:
  - another process (same user/ admin) // syscall kill(*pid*, *n*)
  - kernel
  - *P* itself
- When *P* gets a signal *n*, it executes a "signal handler", say *sh*
  - signal *n* is pending until *P* starts executing *sh*
  - for each *n*, at most one signal *n* can be pending at *P*
  - at any time, *P* can be executing at most one signal handler
- Each *n* has a default handler: ignore signal, terminate *P*, ...
- *P* can register handlers for some signals // syscall signal(*sh*, *n*)
  - if so, *P* also registers a trampoline function,
    which issues syscall complete_handler

Signals: implementation

- *P*'s pcb has
  - *pending* bit for each *n* // true iff signal *n* pending
  - *ongoing* bit // true iff any signal handler is being executed
- When *P* gets a signal *n*, kernel sets *pending n*.
  Causes *sh* to execute at some point when *P* is not running
- When kernel-handled *pending n* and not *ongoing*:
  - kernel sets *ongoing*, clears *pending n*, starts executing its *sh*
  - when *sh* ends, kernel unsets *ongoing*.
- When user-handled *pending n*, not ongoing, and *P* in user mode:
  - kernel sets *ongoing*, clears *pending n*,
    saves *P*'s stack(s) somewhere and modifies them so that
    - *P* will enter *sh* with argument *n*
    - *P* will return from *sh* and enter trampoline
  - when *P* returns to kernel (via complete_handler),
    kernel clears *ongoing* and restores *P*'s stack(s)

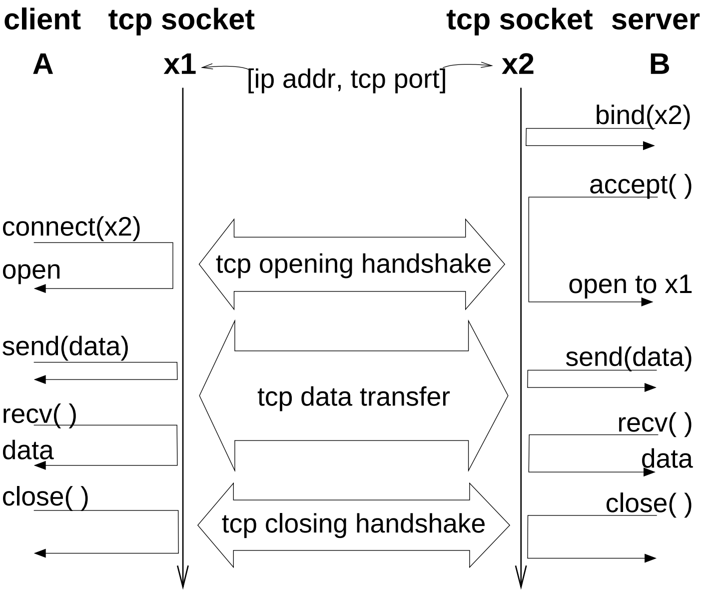
Stacks when handling user-level signal (x86 style)

user stack	**kernel stack**	
ustack0	istate0 usp0	prior to resuming *P* in user mode, signal *n* pending - istate0: interrupt state of process *P* - usp0: top of user stack
ustack0 *n* trampoline	istate1 usp1	prior to resuming *P* at *sh* in user mode - istate1: istate0 with eip ← sh - usp1: usp0 − sizeof(n, &trampoline)
ustack0 *n*	istate2 usp2	just after executing syscall complete_handler
ustack0	istate0 usp0	just prior to resuming *P* at istate0 - istate0 and usp0 restored

Outline

Internet Streaming Sockets

- Two-way data path: client process ↔ server process
- Server:
  - ss ← socket`(INET, STREAMING)` // get a socket
  - bind(ss, server port)
  - client addr:port ← accept(ss)
  - send(ss, data) // byte stream
  - data ← recv(ss) // byte stream
  - close(ss) // returns when remote also closes
- Client
  - sc ← socket`(INET, STREAMING)` // get a socket
  - status ← connect(sc, server addr:port) // returns sucess or fail
  - send(sc, data) // byte stream
  - data ← recv(sc) // byte stream
  - close(sc)



client **tcp socket** **tcp socket** **server**

A **x1** [ip addr, tcp port] **x2** **B**

bind(x2)

accept( )

connect(x2)

open

tcp opening handshake

open to x1

send(data)

tcp data transfer

send(data)

recv( )

data

recv( )

data

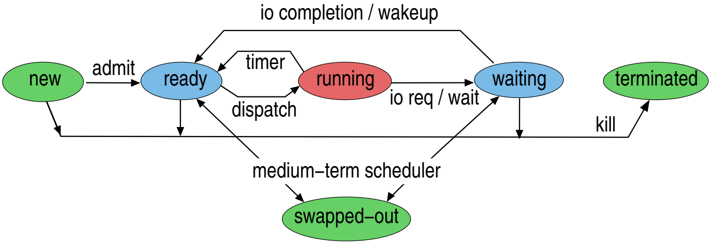
close( )

tcp closing handshake

close( )

Outline

Schedulers



- Short-term (milliseconds) : ready → running
  - high utilization: fraction of time processor doing useful work
  - low wait-time: time spent in ready queue per process
  - fairness / responsiveness: wait-time vs processor time

- Medium-term (seconds): ready/waiting ↔ swapped-out
  - avoid bottleneck processor/device (eg, thrashing)
  - ensure fairness
  - not relevant for single-user systems (eg, laptops, workstations)

Short-term: Non-Preemptive

- Non-preemptive: running $\not\to$ ready
- Wait-time of a process: time it spends in ready queue

- FIFO
  - arrival joins at tail // from waiting, new or suspended
  - departure leaves from head // to running
  - favors long processes over short ones
  - favors processor-bound over io-bound
  - high wait-time: short process stuck behind long process

- Shortest-Job-First (SJF)
  - assumes processor times of ready PCBs are known
  - departure is one with smallest processor time
  - minimizes wait-time

- Fixed-priority for processes: eg: system, foreground, background

Short-term: Preemptive – 1

- Preemptive: running ⟶ ready
- Wait-time of a process: total time it spends in ready queue

- Round-Robin
  - FIFO with time-slice preemption of running process
  - arrival from running, waiting, new or suspended
  - all processes get same rate of service
  - overhead increases with decreasing timeslice
  - ideal: timeslice slightly greater than typical cpu burst

Short-term: Preemptive – 2

- Multi-level Feedback Queue
  - priority of a process depends on its history
  - decreases with accumulated processor time

  - queue 1, 2, $\cdots$, queue N // decreasing priority
  - departure comes from highest-priority non-empty queue
  - arrival coming not from running:
    - joins queue 1
  - arrival coming from running
    - joins queue $\min(i+1, N)$ // i was arrival's previous level

  - To avoid starvation of long processes
    - longer timeslice for lower-priority queues
    - after a process spends a specified time in low-priority queue move it to a higher-priority queue
    - ...

Multiprocessor Scheduling

- Set of ready processes is shared
- So scheduling involves
  - get lock on ready queue
    - ensure it is not in a remote processor's cache
  - choose a process (based on its usage of processor, resources, ...)

- Process may acquire affinity to a processor (ie, to its cache)
  - makes sense to respect this affinity when scheduling
- Per-processor ready queues simplifies scheduling, ensures affinity
  - but risk of unfairness and load imbalance
- Could dedicate some processors to long-running processes and others to short/interactive processes