Overview
Process State
Process Creation
Process Termination
User-Threads Management
Booting the OS
Inter-Process Communication: Pipes
Inter-Process Communication: Signals
Inter-Process Communication: Internet Sockets
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 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 (e.g., memory, disk)
    - memory vs disk (swapped out)
  - user thread status: running, ready, waiting, mode
- Kernel thread: kernel-stack, processor state

- Schedulers:
  - short-term: ready $\rightarrow$ running
  - io device: waiting $\rightarrow$ io service $\rightarrow$ ready
  - medium-term: ready/waiting $\leftrightarrow$ swapped-out
  - long-term: start $\rightarrow$ ready
  - efficiency and responsiveness
Outline

Overview
Process State
Process Creation
Process Termination
User-Threads Management
Booting the OS
Inter-Process Communication: Pipes
Inter-Process Communication: Signals
Inter-Process Communication: Internet Sockets
Schedulers
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
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
Create Process (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
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

Overview
Process State
Process Creation
Process Termination
User-Threads Management
Booting the OS
Inter-Process Communication: Pipes
Inter-Process Communication: Signals
Inter-Process Communication: Internet Sockets
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 (if any)
  - becomes a root process’s child (orphan) // Unix
  - is terminated // VMS
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, ...
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
Overview
Process State
Process Creation
Process Termination
User-Threads Management
Booting the OS
Inter-Process Communication: Pipes
Inter-Process Communication: Signals
Inter-Process Communication: Internet Sockets
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 need no console
    - idle
    - io-servers
  - login/shell process bound to console
  - mount filesystem(s) in io device(s)
Outline

Overview
Process State
Process Creation
Process Termination
User-Threads Management
Booting the OS
Inter-Process Communication: Pipes
Inter-Process Communication: Signals
Inter-Process Communication: Internet Sockets
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
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();
```
Opening a pipe

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

```c
text
```
```c
int pfd[2];
pipe(pfd);
fork();
```
Example: data transfer on pipe from parent to child

- Enter a command, say `prog1`, in shell
- Shell forks-executes a process, say `A`, executing `prog1`
- `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
  - If `B` has closed read-end: ?
- `B` blocks if buffer is empty and `A` has not closed write-end
  - If `A` has closed write-end: ?
Overview
Process State
Process Creation
Process Termination
User-Threads Management
Booting the OS
Inter-Process Communication: Pipes
Inter-Process Communication: Signals
Inter-Process Communication: Internet Sockets
Schedulers
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 is a signal handler being executed

- When P gets a signal n, kernel sets pending n

- 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, unsets pending n, saves P’s stack(s), 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 unsets ongoing and restores P’s stack(s)
### Stacks when handling user-level signal (x86 style)

<table>
<thead>
<tr>
<th>user stack</th>
<th>kernel stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>ustack0</td>
<td>istate0</td>
</tr>
<tr>
<td>usp0</td>
<td></td>
</tr>
</tbody>
</table>

prior to resuming $P$ in user mode, signal $n$ pending
- istate0: interrupt state of process $P$
- usp0: top of user stack

| ustack0     | istate1      |
| n           | usp1         |
| trampoline  |              |

prior to resuming $P$ at $sh$ in user mode
- istate1: istate0 with eip $\leftarrow$ sh
- usp1: usp0 $-$ sizeof(n, &trampoline)

| ustack0     | istate2      |
| n           | usp2         |

just after executing syscall `complete_handler`

| ustack0     | istate0      |
| n           | usp0         |

just prior to resuming $P$ at istate0
- istate0 and usp0 restored
Outline

Overview
Process State
Process Creation
Process Termination
User-Threads Management
Booting the OS
Inter-Process Communication: Pipes
Inter-Process Communication: Signals
Inter-Process Communication: Internet Sockets
Schedulers
Internet Streaming Sockets

- Two-way data path: client process ↔ server process

**Server:**
- \( \text{ss} \leftarrow \text{socket(INET, STREAMING)} \)  // get a socket
- \( \text{bind} \) (ss, server port)
- client addr:port \( \leftarrow \text{accept} \) (ss)
- \( \text{send} \) (ss, data)  // byte stream
- data \( \leftarrow \text{recv} \) (ss)  // byte stream
- close (ss)  // returns when remote also closes

**Client**
- \( \text{sc} \leftarrow \text{socket(INET, STREAMING)} \)  // get a socket
- status \( \leftarrow \text{connect} \) (sc, server addr:port)  // returns success or fail
- \( \text{send} \) (sc, data)  // byte stream
- data \( \leftarrow \text{recv} \) (sc)  // byte stream
- close (sc)
A client socket connects to a server socket through a TCP connection. The process includes:

1. Client (A) opens a socket.
2. Client (A) connects to server (B) via open to x1.
3. Server (B) binds and listens for connections.
4. Server (B) accepts a connection from client (A).
5. Client (A) sends data.
6. Server (B) receives data.
7. Server (B) sends data.
8. Client (A) receives data.
9. Client (A) closes the connection.
10. Server (B) closes the connection.

TCP opening handshake and data transfer are also depicted.
Outline

Overview
Process State
Process Creation
Process Termination
User-Threads Management
Booting the OS
Inter-Process Communication: Pipes
Inter-Process Communication: Signals
Inter-Process Communication: Internet Sockets
Schedulers
Schedulers

- Short-term (milliseconds): ready process → running
- Medium-term (seconds): ready/waiting process ↔ suspended
- Long-term schedule (minutes): new process → ready / suspended

Goals of medium/long term scheduling
- avoid bottleneck processor/device (eg, thrashing)
- ensure fairness
- not relevant for single-user systems (eg, laptops, workstations)

Goals for short-term scheduling
- high utilization: fraction of time processor doing useful work
- high throughput: # processes completed / unit time
- low wait-time: time spent in ready queue / process
- fairness / responsiveness: wait-time vs processor time
- favor high-priority, static vs dynamic // priority inversion
Short-term: Non-Preemptive

- Non-preemptive: running $\rightarrow$ 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 $\rightarrow$ 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, ⋯, 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
  - ...

Scheduler
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