Operating Systems: Processes and Threads

Shankar

January 28, 2021
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
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
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
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 \(\rightarrow\) 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

```
new  admit  ready  timer  running  io completion / wakeup
  |                  | dispatch  | io req / wait
  |                  |          | waiting
  |                  |          | io completion / wakeup
  |                  |          | io req / wait
  |                  |          | waiting
  |                  |          | medium-term scheduler
  |                  |          | swapped-out
  |                  |          | killed
```
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

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
Approach 1: Create Process from Scratch

- CreateProcess(*path*, *context*):
  - read file from file system’s *path* // GeekOS Spawn // executable file // code, data, stack(s), ...
  - acquire memory segments
  - unpack file into its segments
  - create PCB
  - update PCB with *context* // pid, ...
  - add PCB to ready queue // 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

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
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, ...
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
- `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
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 $\rightarrow$ 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)
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
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);
```

- **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()`

```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);
```

### Open file table

<table>
<thead>
<tr>
<th>FD</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>open file (read)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>position</td>
<td>n/a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ref. count</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>open file (write)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>position</td>
<td>n/a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ref. count</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Inode table entry

- **Permissions**: 0666
- **Size**: 0
- **Type**: pipe

© 2016 L. Herman & A. U. Shankar
After a `fork()`

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

**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

- enter a command, say `prog1`, in shell
- shell forks-exec 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: ?
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
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 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)
Stages 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 process $P$ in user mode, signal $n$ pending:
- $istate0$: interrupt state of process $P$
- $usp0$: top of user stack

<table>
<thead>
<tr>
<th>User Stack</th>
<th>Kernel Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>ustack0</td>
<td>istate1</td>
</tr>
<tr>
<td>n</td>
<td>usp1</td>
</tr>
<tr>
<td>trampoline</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>User Stack</th>
<th>Kernel Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>ustack0</td>
<td>istate2</td>
</tr>
<tr>
<td>n</td>
<td>usp2</td>
</tr>
</tbody>
</table>

Just after executing syscall `complete_handler`:

<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>

Just prior to resuming process $P$ at $istate0$:
- $istate0$ and $usp0$ restored
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
Internet Streaming Sockets

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

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

Client
- $sc \leftarrow \text{socket}(\text{INET, STREAMING})$ // get a socket
- $\text{status} \leftarrow \text{connect}(sc, \text{server addr:port})$ // returns success or fail
- $\text{send}(sc, \text{data})$ // byte stream
- $\text{data} \leftarrow \text{recv}(sc)$ // byte stream
- $\text{close}(sc)$
client tcp socket
A x1 [ip addr, tcp port] x2

<table>
<thead>
<tr>
<th>open to x1</th>
<th>accept( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect(x2)</td>
<td>bind(x2)</td>
</tr>
<tr>
<td>open</td>
<td>open to x1</td>
</tr>
<tr>
<td>send(data)</td>
<td>send(data)</td>
</tr>
<tr>
<td>recv()</td>
<td>recv()</td>
</tr>
<tr>
<td>data</td>
<td>data</td>
</tr>
<tr>
<td>close()</td>
<td>close()</td>
</tr>
</tbody>
</table>

TCP opening handshake
TCP data transfer
TCP closing handshake
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
Schedulers

- Short-term (milliseconds): ready process $\rightarrow$ running
- Medium-term (seconds): ready/waiting process $\leftrightarrow$ suspended
- Long-term schedule (minutes): new process $\rightarrow$ 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
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
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

...
- 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