Lecture 2 - Processes & Threads

Yuchen Ouyang

Outline

- process abstraction
- thread abstraction
- process management in Linux
- Paper - lightweight contexts
- Paper - scheduler activation

1 Process Abstraction

1.1 Process is a running executable file

process = a running executable file/program.

The executable file includes:

- executable **code** / CPU instructions
- **data** used by the instructions

1.2 Process composition

Process is the basic isolated protection unit provided by OS, which includes:

- unique ID (`pid`)
- memory image: stack, heap, code, data, ...
- CPU context: registers (stack pointer, program counter, ...)
- kernel resources (open file table, signal handlers, ...)
- threads

2 Thread Abstraction

The basic execution unit provided by OS, or, a sequence of CPU instructions that are runnable on one processor.

A process can contain one or multiple threads. Every thread is bounded to a certain process. Every time user create a process, OS will create a corresponding thread bounded to the new process.

Thread is the basic unit of scheduling.

2.1 Thread composition

Different:

- thread ID (`tid`, unique in the same process)
- registers (`PC`, `SP`, ...)
- **stack (threads allocate local data and call functions independently)**

Shared in one process:

- heap, text and static data segments
- kernel resources (fd table, ...)

2.2 Process vs. Thread

Threads separate the execution concept from processes. Processes provide the running environment for threads: `pid`, address space, fd table, cwd, ...

Benefits:

- Concurrent program implemented by multiple threads within the same process
- Efficient resource sharing (no extra overhead to create/switch vm and other kernel info)
- Utilization of multi-core architectures
- Improves program structure (an extra abstraction layer, process → threads)

2.3 Kernel threads

User threads can transition to kernel threads by getting into traps (syscall/interrupt/fault/...).

Kernel threads perform the background operations for user threads in the kernel space, who are also managed and scheduled by OS itself. All kernel threads share the same address space.

Kernel maintains a thread table and a process table in the kernel space.

usage:

- CPU load balancing `migration`
- work queue `kworker`

2.4 User threads

User threads are maintained and scheduled by the runtime library in the user space.

invisible to OS, no thread table in the kernel space

User program calls the thread functions instead of involving kernel operations.

ad. portable (api stable for different OSes), small, fast for creation and switch, flexible for needs

2.5 Threading model

1:1

one user thread - one kernel thread

popular

N:1

many user threads - one kernel thread

M:N

many user threads - several kernel threads

3 Linux Process Management

3.1 Process states

- running
- ready
- blocked

"running" - scheduler - "ready"
IO operation: "running" → "blocked"
IO finish: "blocked" → "ready"

3.2 fork - create a copy process

`fork` system call will create a process that is exactly the same as the parent process (except `pid`)

one call - two returns (different return values for parent & child)

OS will copy the page table directly, using cow mechanism to delay actual copy on usage (marking read-only, page faults, handler) → **2 processes have their own separate and identical address space.**

OS will also copy the other states like fd table (`stdin`, `stdout`, `stderr`, ...) and signal handlers.

3.3 exec - execute new program

`exec` system call will call loader to load the executable file in the current context.

`exec` will never return if success

`exec` will replace the stack, heap, data and code by the specified file (new program) and start running the first instruction of the new program.

`pid` and stand io will not be replaced.

Combine `fork` and `exec` can add more flexibility to process control, like change permission, IO redirection, change directory ...

3.4 wait - sync & reap

Parent process can call `wait` to turn into blocked and wait for child processes to terminate.

When any child process call `exit` or have error to be terminated, they wake up the waiting parent process.

Every terminated process becomes a zombie. Its resource will not be reaped until its parent process `wait` for the child. If the parent terminates before any child, the child will be reaped by the init process (`pid = 1`, always runnable, no parent)

3.5 Linux process descriptor `task_struct`

Task → process & thread

Each task is a structure, which contains:

- thread info
- task state (running, ready, blocked . . .)
- kernel stack address
- priority
- parent task
- children tasks
- memory mapping sections
- open files
- signal handlers
- . . .

`fork()` will copy the total struct (3.5 KB) for the new process.

Threads are treated as sharing processes in Linux. They have their `task_struct`

3.6 Threads in Linux

`clone()` can be used for creating threads. The flag arguments direct the clone behavior between processes/threads:

- copy vm space
- copy file table
- copy file system info
- copy signal handlers
- . . .

The first thread (main thread) is the thread group leader.

`exec()` will terminate all threads except the main thread, and the new program is executed on the previous main thread.

4 Paper - Lightweight Contexts

4.1 Fork overhead

fork will copy:

- threads, CPU registers
- memory space
- files, credentials, . . .

`fork()` overhead increases exponentially as the number of `fork()` calls.

4.2 LWC abstraction

(similar resource model in the kernel space)

Light-weight contexts (lwc) **isolate the system resources within the process**, just like threads isolate execution contexts from the process concept.

lwc encapsulates:

- memory space (`mmap`)
- credentials
- files
- CPU registers

Threads are orthogonal to lwcs. Creating an lwc doesn't start running: When a thread switches to it, lwc copies the thread state and start executing.

Every thread can only visit one lwc section, which is isolated from other lwcs.

4.3 Use cases

- snapshot - save the current context and create a new one, possibly restore
- server event-handling isolation - prevent info leaking among different sessions
- isolation of sensitive data - signing data

4.4 Evaluation

Switching cost (˜2ms) is lower than kernel threads and processes (˜4.3ms). (user space context)

5 Paper - Scheduler Activation

5.1 Goals

- performance and flexibility of user threads
  - scheduling policy and concurrency models in the user lib
- functionality of kernel threads
  - no idle processors
  - priority management
  - process switching, change address space

5.2 SA model

kernel-user **notification** model: User runtime can adjust the scheduling mechanism according to the upcall messages from kernel.

M:N threading model

```
M processors --- kernel(OS) ---- runtime(usr) --- N user threads
                                 ->
                            **upcall (signal)**
                                 <-
                              **syscall**
```

require 3 stacks: user thread stack, kernel thread stack, SA stack

Kernel notifies the user-level thread system whenever the kernel changes the number of processors attached to it or the state of a running user thread is changed; User runtime notifies the kernel when the application requires more or fewer processors.

5.2.1 Upcall: kernel → user

- add processors → execute a runnable user thread
- processor preempted → user thread returns to the ready list
- SA blocked → give up the processor
- SA unblocked → user thread returns to the ready list

5.2.2 System call: user → kernel

- add processors → allocate more processors
- idle processor → preempt the processor if another process needs it
- no user thread operation need to be reported to the kernel

5.3 Performance

threads $\geq$ SA » kernel threads $>$ processes

slightly worse

- improve performance or increase granularity of service?

  reduce the number of upcalls (switch back the preempted threads in the critical section, not upcall for preempt threads, . . .)