Operating Systems: Processes and Threads

keleher, shankar

February 13, 2024

Outline

1. Kernel threads
2. User processes
3. Inter-Process Communication: Signals
4. Inter-Process Communication: Internet Sockets
5. Schedulers

Kernel threads: state and queues

- state of a kernel thread:
  - Kernel_Thread struct + stack page
- struct Kernel_Thread:
  - esp, *stackPage, *userContext
  - link for s_allThreadList // constant
  - link for current thread queue // runq, waitq, graveyard
  - numTicks, totalTime, priority, pid, joinq, exitcode, owner, ...
- Thread queues
  - s_allThreadList // all threads
  - s_runQueue // ready (aka runnable) threads
  - s_graveyardQueue // ended and to be reaped
  - various waitQueues // mutex, condition, devices, etc
  - *g_currentThreads[MAX_CPUS] // running thread

Starting kernel-only threads

- Start_Kernel_Thread(startfunc, arg, priority, detached, name):
  - Create_Thread:
    get memory for kthread context (struct and stack page)
    init struct: stackPage, esp, numTicks, pid
    add to the all-thread-list
  - Setup_Kernel_Thread:
    configure stack so that upon switching in it executes
    Launch_Thread, then startfunc, then Shutdown_Thread

    ```
    // stack (bottom to top):
    //    startfunc arg, Shutdown_Thread addr, startfunc addr
    //    0 (eflags), KERNEL_CS (cs), Launch_Thread addr (eip)
    //    fake error code, intrpt#, fake gp regs
    //    KERNEL_DS (ds), KERNEL_DS (es), 0 (fs), 0 (gs)
    ```
  - Make thread runnable: add struct to runq

Current thread

- CURRENT_THREAD: // return the thread struct of the caller
  - disable interrupts
  - ct $\leftarrow$ g_currentThreads[GET_CPU_ID]
  - restore interrupts

Outline

1. Kernel threads
2. User processes
3. Inter-Process Communication: Signals
4. Inter-Process Communication: Internet Sockets
5. Schedulers

User process context

- Context of a user process:
  - Kernel_Thread struct + stack page + struct User_Context
- struct User_Context:
  - name[]
  - ldt[2] // code segment, data segment
  - *ldtDescriptor // segment descriptor
  - *memory, size // memory space for process
  - ldtSelector // index into gdt
  - csSelector, dsSelector // index into ldt
  - entryAddr, argBlockAddr, stackPointerAddr
  - *pageDir, *file_descriptor_table[]
  - refCount, mappedRegions, etc

Spawn user process

- Spawn(program, cmd, *kthread, background):
  - read executable file from filesystem // vfs, pfat
  - unpack elf header and content, extract exeFormat // elf
  - mem ← malloc(program maxva + argblock size + stack page)
  - copy program segments into mem space
  - malloc usercontext and set its fields:
    - *memory ← mem
    - ldt, ldt selectors/descriptors
    - entry point, argblock, stack bottom, ...
  - *kthread ← Start_User_Thread(userContext)

Start user thread

- Start_User_Thread(uc, detached):   // "uc" is "usercontext"
  - Create_Thread:
    malloc kthread struct and stack, init, add to all-thread-list
  - Setup_User_Thread:
    point kthread.usercontext to uc
    configure kernel stack as if it was interrupted in user mode
    // stack (bottom to top):
    //   uc.ds (user ss), uc.stackaddr (user esp)
    //   eflags (intrpt on), uc.cs (cs), uc.entryaddr (eip)
    //   errorcode, intrpt#, gp regs except esi   // fake
    //   uc.argblockaddr (esi), uc.ds (ds, es, fs, gs)
    // How is termination handled?
  - Make thread runnable: add struct to runq

Copying between user and kernel spaces

- User_To_Kernel(usercontext, userptr): // kernel addr of useraddr
  return usercontext.memory + userptr

- Copy_From_User(dstInKernel, srcInUser, bufsize):
  ucontext ← CURRENT_THREAD.usercontext
  srcInKernel ← User_To_Kernel(ucontext, srcInUser)
  memcpy(dstInKernel, srcInKernel, bufsize)

- Copy_To_User(dstInUser, srcInKernel, bufsize):
  ucontext ← CURRENT_THREAD.usercontext
  dstInKernel ← User_To_Kernel(ucontext, dstInUser)
  memcpy(dstInKernel, srcInKernel, bufsize)

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)

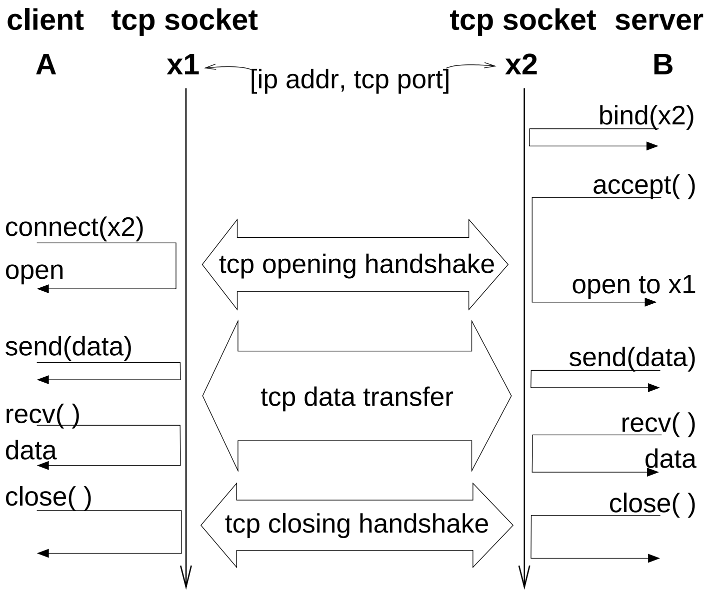
Stacks when handling user-level signal (x86 style)

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

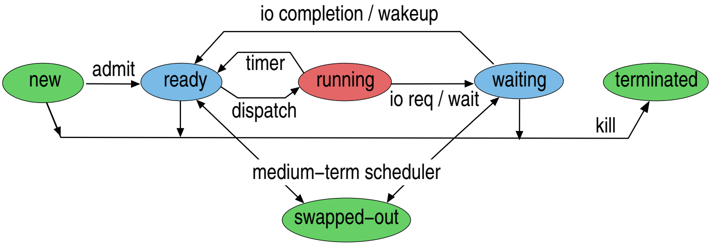
Internet Streaming Sockets

- 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 $\not\rightarrow$ 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 $\longrightarrow$ 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
    - ...