Operating Systems: Processes and Threads

Shankar

March 1, 2015
Outline

Overview

Process State
Process Creation
Process Termination
Schedulers
Booting the OS
Inter-Process Synchronization and Communication
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 and syscalls

- All threads execute concurrently
OS Kernel

Data structure of **current state** of processes + threads
- process: address space, resources, threads
- thread: stack, processor state
- includes mapping of content to hardware location
- thread status: running, ready, waiting, swapped-out

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

Overview
Process State
Process Creation
Process Termination
Schedulers
Booting the OS
Inter-Process Synchronization and Communication
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/stack segments $\rightarrow$ 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
  - address-space: stack
  - status: running, ready, waiting, ...
- ...
- Process swapped-out → all threads swapped out
- User thread:
  - user-mode: when executing user code
  - kernel-mode: when executing system calls
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, ...
  - stack segment
  - 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

- 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
  - eg, to/from environment variables

- User-level vs kernel-level
  - Pro: application-specific scheduling
  - Con: cannot exploit additional processors
Outline

Overview
Process State
Process Creation
Process Termination
Schedulers
Booting the OS
Inter-Process Synchronization and Communication
Approach 1: Create Process from Scratch

CreateProcess(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, ...
// pid, ...
// user, directory, ...
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,
  - 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
  - environment variables are inherited except with exec[lv]e

- For better performance, fork() can
  - copy-on-write, create skeleton segments, ...
Outline

Overview
Process State
Process Creation
Process Termination
Schedulers
Booting the OS
Inter-Process Synchronization and Communication
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, ...
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, \ldots, queue N \quad // \text{decreasing priority}

  - Departure comes from highest-priority non-empty queue
  - Arrival coming not from running:
    - Joins queue 1
  - Arrival coming from running
    - Joins queue \( \min(i + 1, N) \) \quad // \text{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
Outline

Overview
Process State
Process Creation
Process Termination
Schedulers
Booting the OS
Inter-Process Synchronization and Communication
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) on IO device(s)
Outline

Overview
Process State
Process Creation
Process Termination
Schedulers
Booting the OS
Inter-Process Synchronization and Communication
- Process synchronization/communication is essential to OS, eg:
  - multiple threads sharing a buffer
    - at most one thread should modify the buffer at any time
    - a consumer thread must wait if buffer is empty
  - process 1 waits for process 2 to finish
  - process 1 sends data to process 2

- OS typically provides IPSC constructs
  - thread synchronization: locks/conditions, semaphores
  - process synchronization: signals
  - process communication: pipes, sockets

- Next: define constructs and use them to solve generic problems
- Later: implementations (via interrupt disabling, spin locks, ...)

Overview
Atomicity (aka “Mutual Exclusion”)

- Given:
  - code chunks $T_1, \ldots, T_n$
- Obtain “wrapper” $\alpha(T_i)$ st
  - at most one thread is in any $T_i$ at any time
  - thread in $\alpha(T_i)$ returns only after executing $T_i$ (blocking)
  - thread in $\alpha(T_i)$ returns if no thread stays forever in any $T_k$
- Above aka: atomically execute $T_i$

- Typically
  - $T_1$’s are the only way to access some shared structure $S$
  - $T_i$ may not be textually contiguous, eg, have fn calls, intrpts
Conditional atomicity

- Given:
  - code chunks $T_1, \cdots, T_n$
  - boolean expressions $B_1, \cdots, B_n$  // guards

- Obtain “wrapper” $\beta(B_i, T_i)$ st
  - at most one thread in any $T_i$ or $B_i$ at any time
  - thread in $\beta(B_i, T_i)$ executes $T_i$ only if $B_i$ holds, o/w it waits
  - thread in $\beta(B_i, T_i)$ returns if no thread stays forever in any $B_k$ or $T_k$

- Above aka: atomically execute $T_i$ with guard $B_i$

- Typically
  - $T_1$’s, $B_i$’s are the only access some shared structure $S$
  - $B_i$ have so side-effects on $S$
  - some $B_i$’s can be missing  // equivalent to true
  - if all $B_i$’s are missing, reduces to atomicity problem
Outline

Overview
Process State
Process Creation
Process Termination
Schedulers
Booting the OS
Inter-Process Synchronization and Communication
  Locks
  Conditions
  Semaphores
  Signals
  Pipes and Sockets
Lock: Definition

- Lock aka “mutex” in GeekOS/POSIX
- A lock is either acquired by a thread or free

- Lock lck: // initially free
- lck.acq(): // acquire
  // caller does not hold lock
  atomically acquire lck only if free, waiting if not free
  return
- lck.rel(): // release
  // caller holds lock
  free lck
  return

- Progress: a thread at lck.acq() eventually gets past if
  - lock is free continuously // weak lock
  - lock is free continuously or repeatedly // strong lock
Atomicity using Locks

- Consider code chunks $T_1, T_2, \cdots, T_n$
- Introduce a lock `lck`
- Change every $T_k$ to
  
  ```
  lck.acq()
  T_k
  lck.rel()
  ```
Consider code chunks $T_1, \ldots, T_n$ and guards $B_1, \ldots, B_n$

Change $T_k$ to

```
1: lck.acq()
   if (not $B_k$)
     lck.rel()
     short nap  // optional
     goto 1
   else
     $T_k$
     lck.rel()
```

Ugly!
Outline

Overview
Process State
Process Creation
Process Termination
Schedulers
Booting the OS
Inter-Process Synchronization and Communication
  Locks
  Conditions
  Semaphores
  Signals
  Pipes and Sockets
Condition (lck) cv:  // cond cv is associated with lock lck

cv.wait():
  // caller must hold lck
  atomically release lck and wait on cv          // always blocks
  acquire lck
  return

cv.signal():  // aka “wakeup”, never blocks
  // caller must hold lck
  wake up a process (if any) waiting on cv
  return

Progress: a thread at cv.wait() eventually gets past if
  - cv is signalled, and no other process is waiting on cv  // weak
  - cv is repeatedly signalled  // strong
Conditional atomicity using Conditions

- Consider code chunks $T_1, \ldots, T_n$ and guards $B_1, \ldots, B_n$
- Introduce a lock $lck$
- Introduce a conditions $c_1, \ldots, c_n$ assoc with $lck$

- Change $T_k$ to:
  
  ```
  lck.acq()
  while (not $B_k$)
    $c_k$.wait()
  $T_k$
  WakeUpAllPossible
  lck.rel()
  ```

- $WakeUpAllPossible$:  
  ```
  if ($B_1$)
    $c_1$.signal()
  \ldots
  \ldots
  if ($B_n$)
    $c_n$.signal()
  ```

- If a $B_k$ is missing: no need for $c_k$
- If $B_k$ and $B_j$ are the same: $c_k$ suffices; no need for $c_j$
Outline

Overview
Process State
Process Creation
Process Termination
Schedulers
Booting the OS
Inter-Process Synchronization and Communication
  Locks
  Conditions
  Semaphores
  Signals
  Pipes and Sockets
Counting Semaphore: Definition

- Synchronization: mutual exclusion + conditional wait
- Semaphore(N) sem  // semaphore sem initialized to N ≥ 0
- sem.V():  // aka “signal”, “increment”, “release”
  atomically increment sem
  return
- sem.P():  // aka “wait”, “decrement”, “acquire”
  atomically decrement sem only if sem > 0  // blocking
  return
- Progress: when does a thread at P eventually gets past
  - weak: sem > 0 holds continuously
  - strong: sem > 0 continuously or intermittently
- Note: P() need not give waiting threads priority over new arrivals
Binary Semaphore: Definition

- Semaphore that is either 0 or 1
- `BinarySemaphore(N) sem` // `N` is 0 or 1
- `sem.V()`:
  - `atomic {sem ← 1}
  - return`
- `sem.P()`:
  - `atomic {sem ← 0 only if sem=1}
  - return` // blocking
- Progress: same as counting semaphores
- Can implement counting semaphores using binary semaphores
- Binary semaphores may be easier to implement in hardware
Atomicity using Semaphores

- Consider code chunks $T_1, \ldots, T_n$
- Semaphore(1) sem
- Change $T_i$ to
  
  ```
  sem.P()
  T_i
  sem.V()
  ```

- “mutex” is a common name for such semaphores
  
  - note: don’t confuse with “Mutex” construct in GeekOS
Conditional atomicity using Semaphores

- Consider code chunks $T_1, \cdots, T_n$ and guards $B_1, \cdots, B_n$

- Semaphore(1) $mutex$ // for atomicity
- Semaphore(0) $gate_1, \cdots, gate_n$ // for waiting on $B_i$

- Change $T_k$ to:
  
  $mutex.P()$
  while (not $B_k$)
  
  $mutex.V()$
  $gate_k.P()$
  $mutex.P()$

  $T_k$
  $WakeUpAllPossible$
  $mutex.V()$

- $WakeUpAllPossible$:  
  
  if ($B_1$)
  
  $gate_1.V()$
  
  \ldots
  
  \ldots
  
  if ($B_n$)
  
  $gate_n.V()$

- “gate” is a common name for semaphores used in this way
Outline

Overview
Process State
Process Creation
Process Termination
Schedulers
Booting the OS
Inter-Process Synchronization and Communication
  Locks
  Conditions
  Semaphores
  Signals
  Pipes and Sockets
Signal: process-level interrupt; issued by OS or process
- SIGKILL, SIGALRM, SIGCHILD, SIGSTOP, SIGCONT, ...
- SIGBUS, SIGSEGV, SIGFPE, SIGILL, SIGPIPE, ... // self-caused

Process has signal-handler functions
- default signal handler
- register signal handler for a signal // via a syscall

Signal protocol
- kernel/process sends a signal to a process // via a syscall
- target process eventually executes the signal handler
- asynchronous; blockable
When a signal is sent to process A
  - OS notes this in A’s PCB

When A is to be dispatched
  - if A’s PCB has no pending signal
    - A resumes from the point, say X, where it was switched out
  - if A’s PCB has a pending signal, OS sets up A’s stack so that
    - A “resumes” to signal handler (instead of to X)
    - A returns from signal handler into X
Overview
Process State
Process Creation
Process Termination
Schedulers
Booting the OS
Inter-Process Synchronization and Communication
  Locks
  Conditions
  Semaphores
  Signals
Pipes and Sockets
■ Fifo buffer from a process A to a process B
  ■ Pipe, A, B are created atomically (eg, by a shell)

■ Eg: shell command “p1 | p2” causes the shell to
  ■ create a pipe (fifo buffer)
  ■ start a process executing p1, with write access to pipe
  ■ start a process executing p2, with read access to pipe
  ■ byte stream: in-chunks need not equal out-chunks
  ■ p1 (p2) blocks if buffer is full (empty)       // synchronization
  ■ one side closes → SIGPIPE signal to the other side
Unix/Internet Streaming Sockets

- Two-way data path: server process ↔ client process(es)

- Server:
  - server-location: path / addr-port  // Unix / Internet socket
  - socket(unix/inet): returns socket, say ss
  - bind(ss, server-location)
  - listen(ss,)
  - accept(ss): returns another socket, say s2
  - send(s2,), recv(s2)  // byte stream
  - close(s2)

- Client
  - socket(unix/inet): returns socket, say sc
  - connect(sc, server-location)
  - send(sc,), recv(sc)  // byte stream
  - close(sc)