Seongwoo Choi
CMPS 111 - 01
Winter 2017
Week 3 Note

Processes and Threads

Flex is a tool that takes a regular expression and put it into a C code. More detail can be discovered at gnu.org. This is a public organization website that talks about how GNU affected modern open source project.

Shell is a simple language. Professor Long briefly explained about how to approach shell program. Also, he mentioned that students should not upload any format of file that was pirated from the internet.

What is a process?

Code, data, and stack

Usually (but not always) has its own address space.

Program state.

CPU registers

Program counter (current location in the code)

Stack pointer

Only one process can be running in a single CPU core at any give time!

Multi-core CPUs can support multiple processes.

Return address goes to stack

Kernels have stacks
Each process has stack.

Note: Professor Long answered a phone call from his father during the class.

When is a process created?

Processes can be created in two ways

System initialization: one or more processes created when the OS starts up.

Execution of a process creation system call something explicitly asks for a new process.

System calls can come from

User request to create a new process (system call executed from user shell)

Already running processes

User programs
System daemons

When do processes end?

Conditions that terminate processes can be

- Voluntary
- Involuntary

Voluntary

- Normal exit
- Error exit

Involuntary

- Fatal error (only sort of involuntary)
- Killed by another process

Process Hierarchies

Parent creates a child process

- Child processes can create their own children

Forms a hierarchy

- UNIX calls this a "process group"
- If a process terminates, its children are "inherited" by the terminating process's parent.

Windows has process groups

- Multiple processes grouped together
- One process is the "group leader"

Process in one of 5 states

- Created
- Ready
- Running
- Blocked
- Exit

Transitions between states

Process enters ready queue.

Scheduler picks this process

Scheduler picks a different process

Processes in the OS.

- Two "layers" for processes
- Lowest layer of process-structured OS handles interrupts, scheduling.

Above that layer are sequential processes.

- Processes tracked in the process table.

Each process has a process table entry.

Processes

0 1 … N-2 N-1

Scheduler

What's in a process table entry?

Process management. Registers Program counter CPU status word Stack pointer Process state Priority/scheduling Parameters Process ID Parent process ID Signals Process start time Total CPU time	**File management** Root directory Working directory File descriptors User ID Group ID **Memory management** Pointers to text, data, stack or Pointers to page. Table.

What happens on a trap / interrupt?

Hardware saves program counter (on stack or in a special register)

Hardware loads new PC, identifies interrupt

Assembly language routine saves registers

Assembly language routine sets up stack.

Assembly language calls (to run service routine)

Service routine calls scheduler

Scheduler selects a process to run next (might be the one interrupted)

Assembly language routine loads PC & registers for the selected process.

Threads "Processes" sharing memory

Process ⇔ address space

Thread ⇔ program counter / stream of instructions

Two examples

- Three processes, each with one thread
- One process with three threads

Process and thread information

Per process items Address space Open files Child processes Signals & handlers Accounting info Global variables		
Per thread items Program counter Registers Stack & stack pointer State (local variables)	**Per thread items** Program counter Registers Stack & stack pointer State (local variables)	**Per thread items** Program counter Registers Stack & stack pointer State (local variables)

Threads: "processes" sharing memory

Process ⇔ address space

Two examples

Three processes, each with one thread.

One process with three threads.

Dijkstra said "goto is bad". Goto considered harmful. It is better to use 'return' command on your code.

What's in a process

Why use threads?

Allow a single application to do many things at once
- Simpler programming model
- Less waiting

Threads are faster to create or destroy
- No separate address space

Overlap computation and I/O
- Could be done without threads, but it's harder

Example: word processor
- Thread to read from keyboard
- Thread to format document
- Thread to write to disk

Three ways to build a server

Multithreaded server

- Parallelism
- Blocking system calls
- May use pop-up threads: create a new thread in response to an incoming message (rather than reusing a thread)

Single-threaded process: slow, but easier to do

- No parallelism
- Blocking system calls

Finite-state machine (event model)

- Each activity has its own state: states change when system calls complete or interrupts occur
- Parallelism
- Nonblocking system calls

Issues with using threads

There are some issues that involve with using threads. It may be tricky to convert single-threaded code to multi-threaded code.

Re-entrant code

- Code must function properly when multiple threads are using it simultaneously
- Need to be careful when using static or global variables
- Returned structures
- Buffers

Error management

- What happens when just a single thread has an error?

This is the reason why there is an error management. However, we cannot simply kill the process because there might be other threads running in the background.

Professor Long mentioned that
Global variables are bad and we have to make codes pure.

Every time you do something, you need to use kernel.
What do I have?
One thread will call right, another thing will go to sleep
Schedule threads will solve the problem.

POSIX threads
Standard interface to threading library
May be implemented in either user or kernel space
Some operating systems provide support for both!

Allows thread-based programs to be portable

Processes & threads in Linux

Supports POSIX standard
Linux supports kernel-level threads (lightweight processes)
• Share address space, file descriptors, etc.
• Each has its own process descriptor in memory
Linux processes (incl. lightweight) all have unique identifiers
• Threads sharing address space are grouped into process groups
• Identifier shared by the group is that of the leader
Each process has its own 8KB region that stores
• Kernel stack
• Kernel has a small stack: about 4KB!
• Low-level thread information
Other information stored in a separate data structure
Memory allocated to the process
• Open files
• Signal information

Assignment 2
Scheduling - It is just like lottery.
Lottery scheduling

6:15PM

Professor changed to other presentation.

Multicore Operating Systems

"The way the processor industry is going is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it.

Why does multicore matter?

Nowadays, Intel or any other processor manufacturers declared that the gigahertz wars are over: clock rates haven't gone up in years.
So how can CPUs get faster? Like what are the differences between the last year's model and this year's model? How do they make them faster?
There are two techniques that companies do:
  Better microarchitecture: more instructions per cycle
  More CPU cores: multiprocessor on a chip.
Sometimes threads share a core: hyper threading.

Even cell phones and tablets have multi core CPUs
They are usually:
Feature size: 3 nm
CPU runs at three gigahertz
Examples are snapdragon, Apple A9 chip, Samsung Exynos, and etc.

Multicore architecture

Multiple CPU cores
Each core supports 1 or more independent threads
Multiple cache levels
L1 cache typically “bound” to single core
L2 cache shared by multiple cores
L3 cache shared by all cores (usually)
Access to “local” data is faster
Data access is complex
Caches need to be kept consistent
Written values may need to be copied to a new cache

90% of CPU time happens only because of 10% of codes.
Processes have a font print_in L1, L2
When content switch, drop L1 cache.

What makes multicore OS harder?

Synchronization
CPUs need to use synchronization primitives
Multiple cores are independent
Can’t disable interrupts: each core has its own interrupts

Need to use test-and-set or compare-and-swap instruction
Single instruction locks the memory bus
No other core can use the bus
Spin locks become very expensive

Complicated by memory design
Doing atomic instruction on data in someone else’s cache is expensive
How can multiple cores synchronize efficiently?

Dutch mathematician Dekker and Peterson’s algorithm were mentioned during the class.

Spin or switch?
Spin locks slow all cores
Repeated bus locking
Should the waiting core spin or switch threads?

Spin: avoid cost of switching threads
Good if the spin lock won't be held long

Switch: avoid cost spinning
Good if the lock might be held longer
Good if a switch was going to happen soon anyway

CPU-based solution for multi core spin locks
Best effort scheduling software scheduling

Scheduling
Why schedule processes?
Bursts of CPU usage alternate with periods of I/O wait
Some processes are CPU-bound: they don't many I/O requests
Other processes are I/O-bound and make many kernel requests

Motivation is different but idea is same.

When are processes scheduled? quanta
At the time they enter the system
Common in batch systems
Two types of batch scheduling
Submission of a new job causes the scheduler to run
Scheduling only done when a job voluntarily gives up the CPU (i.e., while waiting for an I/O request)
At relatively fixed intervals (clock interrupts)
Necessary for interactive systems
May also be used for batch systems
Scheduling algorithms at each interrupt, and picks the next process from the pool of "ready" processes.