DISTRIBUTED COMPUTER
SYSTEMS

PROCESSES AND THREADS

Dr. Jack Lange
Computer Science Department
University of Pittsburgh

Fall 2015

Outline

- Heavy Weight Processes
- Threads and Thread Implementation
  - User Level Threads
  - Kernel Level Threads
  - Light Weight Processes
  - Scheduler Activation
- Client/Server Implementation
  - Multithreaded Server
  - Xwindow
  - Server General Design Issues
Process and Process Management

PROCESS CONCEPT

Process Concept

A primary task of an operating system is to execute and manage processes.

What is a program and what is a process?

- A program is a specification, which contains data type definitions, how data can be accessed, and set of instructions that when executed accomplishes useful “work”.
- A sequential process is the activity resulting from the execution of a program with its data by a sequential processor.

- Program is passive, while a process is active.

A process is an instance of a program in execution.
Process Traditional Structure

- Address Space
  - Code – Binary code
  - Static and dynamic data
  - Execution stack – Contains data such as subroutine parameter, return address, and temporary variables

- Process CPU State
  - Process Status Word (PSW) – Execution mode, Last operation outcome, Interrupt level, …
  - Instruction Register (IR) – Current instruction being executed
  - Program Counter (PC) – Next instruction to be executed
  - Stack pointer (SP)
  - General purpose registers

Process and Process Management

**PROCESS ADDRESS SPACE**
Memory Layout

32-bit Linux Kernel/User Memory Split

32-bit Windows Default Memory Split

Process Address Space

<table>
<thead>
<tr>
<th>Name</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Segment</td>
<td>Program executable statements</td>
</tr>
<tr>
<td>Data Segment</td>
<td>1. Static or global data initialized with non-zero values</td>
</tr>
<tr>
<td>2. Zero-Initialized or Block Started by Symbol (BSS) data</td>
<td>2. Uninitialized static or global data</td>
</tr>
<tr>
<td>Stack Segment</td>
<td>Categories occupy a single contiguous area.</td>
</tr>
<tr>
<td>Stack</td>
<td>Local variables of the scope.</td>
</tr>
<tr>
<td>Data</td>
<td>Function parameters</td>
</tr>
<tr>
<td>Heap</td>
<td>Dynamic memory allocation</td>
</tr>
<tr>
<td>Text</td>
<td></td>
</tr>
</tbody>
</table>
Process and Process Management

PROCESS STATE

Process States

- Process in one of 5 states
  - Created
  - Ready
  - Running
  - Blocked
  - Exit

- Transitions between states
  1 - Process enters ready queue
  2 - Scheduler picks this process
  3 - Scheduler picks a different process
  4 - Process waits for event (such as I/O)
  5 - Event occurs
  6 - Process exits
  7 - Process ended by another process
Processes Representation

- To users and to other processes, a process is identified by its unique Process ID (PID)
  - PID <= 30,000 in Unix
- In the OS, processes are represented by entries in a Process Table (PT)
  - PID is index to (or pointer to) a PT entry
  - PT entry = Process Control Block (PCB)
- PCB is a large data structure that contains or points to all information about the process
  - Linux, defined in `task_struct` – It contains about 70 fields
  - NT – defined in `EPROCESS` – It contains about 60 fields

What's In A Process Table Entry?

<table>
<thead>
<tr>
<th>Process management</th>
<th>File management</th>
<th>Memory management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Root directory</td>
<td>Pointers to text, data, stack or Pointer to page table</td>
</tr>
<tr>
<td>Program counter</td>
<td>Working (current) directory</td>
<td></td>
</tr>
<tr>
<td>CPU status word</td>
<td>File descriptors</td>
<td></td>
</tr>
<tr>
<td>Stack pointer</td>
<td>User ID</td>
<td></td>
</tr>
<tr>
<td>Process state</td>
<td>Group ID</td>
<td></td>
</tr>
<tr>
<td>Priority / scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process start time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total CPU usage</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

May be stored on stack
Process Information

<table>
<thead>
<tr>
<th>Process ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>UID GID EUID EGID CWD</td>
</tr>
</tbody>
</table>

Signal Dispatch Table

Memory Map

Priority

Signal Mask

Stack

CPU State

File Descriptors

Process CPU State

- The CPU state is defined by the registers’ contents
  - Process Status Word (PSW)
    - exec. mode, last op. outcome, interrupt level
  - Instruction Register (IR)
    - Current instruction being executed
  - Program counter (PC)
  - Stack pointer (SP)
  - General purpose registers
Process control block (PCB)

<table>
<thead>
<tr>
<th>PCB</th>
<th>kernel</th>
<th>user</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>text</td>
<td>text</td>
<td>PSW</td>
</tr>
<tr>
<td>memory</td>
<td>data</td>
<td>data</td>
<td>IR</td>
</tr>
<tr>
<td>files</td>
<td>heap</td>
<td>heap</td>
<td>PC</td>
</tr>
<tr>
<td>accounting</td>
<td></td>
<td></td>
<td>SP</td>
</tr>
<tr>
<td>priority</td>
<td></td>
<td></td>
<td>general purpose registers</td>
</tr>
<tr>
<td>user</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU registers storage</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operating System Control Structure

Memory

Processes

devices

Files

Memory Table

Process Table

Process 1 PCB

Process 2 PCB

Process n PCB

Device Table

File Table

Process 1 PCB

Process 1 PCB

Process 1 PCB
Process and Process Management

CONTEXT SWITCHING

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

[Diagram of processes and scheduler]

Scheduler
Process Management

- In multi-programming and time sharing environments, process management is required
  - A program may require multiple processes
  - Many processes may be instances of the same program
  - Many processes from different programs can run simultaneously
- How does the OS manage multiple processes running concurrently?
  - What kind of information must be kept?
  - What functions must the OS perform to run processes correctly.

Switching Between Processes

- An important task of the OS is to manage CPU allocation among concurrent processes
  - When the OS takes away the CPU from a running process, the OS must perform a switch between the processes
    - This referred to as context switch
      - It is also referred to as a “state save” and “state restore”
- What is a context?
- What operations must take place to achieve this switch?
- How can the OS guarantee that the interrupted process can resume correctly?
Process Context

- **Process Context** is the necessary information of the current process operational state that must be **stored** in order to later resume process operation from an identical position as when the process was interrupted.

- Context information is system dependent and typically includes:
  - User Level Context
  - Register Level Context
  - System Level Context
    - Address space, stack space, Program Counter (PC), Stack Pointer (SP), Instruction Register (IR), Program Status Word (PSW) and other general processor registers, profiling or accounting information, associated kernel data structures and current state of the process

CPU Switch From Process to Process
Process Context Switch

- Save processor context, including program counter and other registers
- Update the process control block with the new state and any accounting information
- Move process control block to appropriate queue - ready, blocked
- Select another process for execution
- Update the process control block of the process selected
- Update memory-management data structures
- Restore context of selected process and set the PC to the that process code – typically, implicit in the interrupt return instruction

Context Switch Design Issues

- Context can be costly and must be minimized
  - Context switch is purely system overhead, as no “useful” work accomplished during context switching
  - The actual cost depends OS and on the support provided by the hardware
    - The more complex the OS and the PCB -> longer the context switch
    - Some hardware provides multiple sets of registers per CPU, allowing multiple contexts to be loaded at once
    - A “full” process switch may require a significant number of instruction execution.
What Happens On A Trap/Interrupt?

1. Hardware saves program counter (on stack or in a special register)
2. Hardware loads new PC, identifies interrupt
3. Assembly language routine saves registers
4. Assembly language routine sets up stack
5. Assembly language calls C to run service routine
6. Service routine calls scheduler
7. Scheduler selects a process to run next (might be the one interrupted…)
8. Assembly language routine loads PC & registers for the selected process

Process Concept Revisited

Heavy and Light Weight Process Model
Process Characteristics

- In modern operating systems, the two main characteristics of a process are treated separately
  - The *unit of resource ownership* is usually referred to as a *process* or *task*
  - The *unit of execution* is usually referred to a *thread* or a “lightweight process”

Heavyweight Processes

- A *virtual address space* which holds the process image
- *Protected access* to processors, files, and other I/O resources
- Use *message passing* or *shared memory* to communicate with other processes
“Heavyweight” Process Model

- Simple, uni-threaded model
- Security provided by address space boundaries
- High cost for context switch
- Coarse granularity limits degree of concurrency

![Diagram showing User and Kernel Level Threads](image)

Threads

User and Kernel Level Threads
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
Threads & Stacks

Thread 1 | Thread 2 | Thread 3
---|---|---
Thread 1’s stack | Thread 2’s stack | Thread 3’s stack

User space | Process | Kernel

=> Each thread has its own stack!

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
Implementing threads

Kernel
Run-time system
Thread table
Process table

Scheduling User-Level Threads

Kernel picks a process to run next
- Run-time system (at user level) schedules threads
  - Run each thread for less than process quantum
  - For example, a process is allocated 40ms each, threads get 10ms each

- Example schedule: A1,A2,A3,A1,B1,B3,B2,B3
- Not possible: A1,A2,B1,B2,A3,B3,A2,B1
Scheduling Kernel-Level Threads

- Kernel schedules each thread
  - No restrictions on ordering
  - May be more difficult for each process to specify priorities
- Example schedule: A1,A2,A3,A1,B1,B3,B2,B3
- Also possible: A1,A2,B1,B2,A3,B3,A2,B1

ULTs Pros and Cons

**Advantages**
- Thread switching does not involve the kernel – no need for mode switching
  - Fast context switch time,
- Threads semantics are defined by application
- Scheduling can be application specific
  - Best algorithm can be selected
- ULTs are highly portable – Only a thread library is needed

**Disadvantages**
- Most system calls are blocking for processes
  - All threads within a process will be implicitly blocked
  - Waste of resource and decreased performance
- The kernel can only assign processors to processes.
  - Two threads within the same process cannot run simultaneously on two processors
KLT Pros and Cons

**Advantages**
- The kernel can schedule multiple threads of the same process on multiple processors
- Blocking at thread level, not process level
  - If a thread blocks, the CPU can be assigned to another thread in the same process
- Even the kernel routines can be multithreaded

**Disadvantages**
- Thread switching always involves the kernel
  - This means two mode switches per thread switch are required
- KTLs switching is slower compared ULTs
  - Still faster than a full process switch

Combined ULTs and KLTs

Solaris Approach
Combined ULT/KLT Approaches

- Thread creation done in the user space
- Bulk of thread scheduling and synchronization done in user space
- ULT’s mapped onto KLT’s
  - The programmer may adjust the number of KLTs
- KLT’s may be assigned to processors
- Combines the best of both approaches

“Many-to-Many” Model

Solaris Process Structure

- Process includes the user’s address space, stack, and process control block
- **User-level threads** – Threads library
  - Library supports for application parallelism
  - Invisible to the OS
- **Kernel threads**
  - Visible to the kernel
  - Represents a unit that can be dispatched on a processor
- **Lightweight processes** (LWP)
  - Each LWP supports one or more ULTs and maps to exactly one KLT
Solaris Threads

- Task 2 is equivalent to a pure ULT approach – Traditional Unix process structure
- Tasks 1 and 3 map one or more ULT’s onto a fixed number of LWP’s, which in turn map onto KLT’s
- Note how task 3 maps a single ULT to a single LWP bound to a CPU

Solaris – User Level Threads

- Share the execution environment of the task
  - Same address space, instructions, data, file (any thread opens file, all threads can read).
- Can be tied to a LWP or multiplexed over multiple LWPs
- Represented by data structures in address space of the task – but kernel knows about them indirectly via LWPs
Solaris – Kernel Level Threads

- Only objects scheduled within the system
- May be multiplexed on the CPU’s or tied to a specific CPU
- Each LWP is tied to a kernel level thread

Solaris – Versatility

- ULTs can be used when logical parallelism does not need to be supported by hardware parallelism
  - Eliminates mode switching
    - Multiple windows but only one is active at any one time
- If ULT threads can block then more LWPs can be added to avoid blocking the whole application
- High Versatility – System can operate in a Windows style or conventional Unix style, for example
ULTs, LWPs and KLTs

- LWP can be viewed as a virtual CPU
  - The kernel schedules the LWP by scheduling the KLT that it is attached to
- Run-time library (RTL) handles multiple threads handled by RTL
  - When a thread invokes a system call, the associated LWP makes the actual call
    - LWP blocks, along with all the threads tied to the LWP
    - Any other thread in same task will not block.

Thread Implementation

Scheduler Activation
Scheduler Activation – Motivation

- Application has knowledge of the user-level thread state but has little knowledge of or influence over critical kernel-level events – By design, to achieve the virtual machine abstraction
- Kernel has inadequate knowledge of user-level thread state to make optimal scheduling decisions

Underscores the need for a mechanism that facilitates exchange of information between user-level and kernel-level mechanisms.

A general system design problem: communicating information and control across layer boundaries while preserving the inherent advantages of layering, abstraction, and virtualization.

Scheduler Activations: Structure

- Change in Processor Requirements
- Change in Processor Allocation
- Change in Thread Status
Communication via Upcalls

- The kernel-level scheduler activation mechanism communicates with the user-level thread library by a set of upcalls:
  - Add this processor (processor #)
  - Processor has been preempted (preempted activation #, machine state)
  - Scheduler activation has blocked (blocked activation #)
  - Scheduler activation has unblocked (unblocked activation #, machine state)

- The thread library must maintain the association between a thread’s identity and thread’s scheduler activation number.

Role of Scheduler Activations

Invariant: There is one running scheduler activation (SA) for each processor assigned to the user process.
Avoiding Effects of Blocking

Kernel Threads

Scheduler Activations

Resuming Blocked Thread

4: preempt
5: resume
Scheduler Activations – Summary

- Threads implemented entirely in user-level libraries
- Upon calling a blocking system call
  - Kernel makes an up-call to the threads library
- Upon completing a blocking system call
  - Kernel makes an up-call to the threads library
- Is this the best possible solution? Why not?
  - Specifically, what popular principle does it appear to violate?

Servers and Clients
Threads in distributed systems - clients

- Client usage is mainly to hide network latency
- E.g. multithreaded web client
  - Web browser scans incoming HTML page, and finds additional objects to fetch
  - Each file is fetched by a separate thread, each of which performs a blocking I/O operation
  - As files come in, the browser displays them
- Multiple request-response calls to remote systems
  - A client does several RPC calls at the same time, each using a separate thread
  - It waits until all results have been returned
  - Note: if calls are to different servers, we may have a linear speed-up compared to doing calls one after the other

Threads in distributed systems – servers

- In servers, the main issue is improved performance and better structure
- Improve performance:
  - Starting a thread to handle an incoming request is much cheaper than starting a new process
  - Having a single-threaded server prohibits simply scaling the server to a multiprocessor system
  - As with clients: hide network latency by reacting to next request while previous one is being replied
- Better structure:
  - Most servers have high I/O demands. Using simple, well-understood blocking calls simplifies the overall structure.
  - Multithreaded programs tend to be smaller and easier to understand due to simplified flow of control
Multithreaded Web server

while(TRUE) {
    getNextRequest(&buf);
    handoffWork(&buf);
}

while(TRUE) {
    waitForWork(&buf);
    lookForPageInCache(&buf,&page);
    if(pageNotInCache(&page)) {
        readPageFromDisk(&buf,&page);
    }
    returnPage(&page);
}

Multithreaded Servers (1)

- A multithreaded server organized in a dispatcher/worker model.

Dispatcher thread
Request dispatched to a worker thread
Server
Worker thread
Request coming in from the network
Operating system
Three ways to build a server

- Thread model
  - Parallelism
  - Blocking system calls
- Single-threaded process: slow, but easier to do
  - No parallelism
  - Blocking system calls
- Finite-state machine
  - Each activity has its own state
  - States change when system calls complete or interrupts occur
  - Parallelism
  - Non-blocking system calls
  - Interrupts

Multithreaded Servers (2)

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, nonblocking system calls</td>
</tr>
</tbody>
</table>
Out-of-band communication

- How to interrupt a server once it has accepted (or is in the process of accepting) a service request?
  - Solution 1: Use a separate port for urgent data (possibly per service request):
    - Server has a separate thread (or process) waiting for incoming urgent messages
    - When urgent msg comes in, associated request is put on hold
      - Require OS supports high-priority scheduling of specific threads or processes
  - Solution 2: Use out-of-band communication facilities of the transport layer:
    - E.g. TCP allows to send urgent msgs in the same connection
    - Urgent msgs can be caught using OS signaling techniques

Kinds of Servers

- Iterative vs. Concurrent

- Iterative Servers
  - Receive request
  - Perform service
  - Reply if necessary
  - Process next request

- Concurrent Servers
  - Receive request
  - Pass to separate process/thread
  - Receive next request
Servers and state

- Stateless servers: Never keep accurate information about the status of a client after having handled a request:
  - Don’t record whether a file has been opened (simply close it again after access)
  - Don’t promise to invalidate a client’s cache
  - Don’t keep track of your clients

- Consequences:
  - Clients and servers are completely independent
  - State inconsistencies due to client or server crashes are reduced
  - Possible loss of performance because, e.g., a server cannot anticipate client behavior (think of prefetching file blocks)

Servers and state

- Stateful servers: Keeps track of the status of its clients:
  - Record that a file has been opened, so that pre-fetching can be done
  - Knows which data a client has cached, and allows clients to keep local copies of shared data

- Observation: The performance of stateful servers can be extremely high, provided clients are allowed to keep local copies. As it turns out, reliability is not a major problem.
Thin-client computing

- Thin-client
  - Client and server communicate over a network using a remote display control
    - Client sends user input, server returns screen updates
    - Graphical display can be virtualized and served to a client
    - Application logic is executed on the server

- Technology enablers
  - Improvements in network bandwidth, cost and ubiquity
  - High total cost of ownership for desktop computing

The X-Window System

- The basic organization of the X Window System
- Position of window manager affects/reflects the sophistication of the “client” (running the X-“server”)

![X-Window System Diagram]

Server machine
- Application
- Xlib
- Xlib interface
- X protocol
- Terminal (includes display keyboard, mouse, etc.)

Client machine
- X kernel
- Device drivers
Servers: General Design Issues

a) Client-to-server binding using a **daemon** as in Distributed Computing Environment (DCE)

b) Client-to-server binding using a **superserver** as in UNIX

---

Client-Side Software for Distribution Transparency

- A possible approach to transparent replication of a remote object using a client-side solution.
Conclusion – Server Activation

- Heavy Weight Processes
- Threads and Thread Implementation
  - User Level Threads
  - Kernel Level Threads
  - Light Weight Processes
  - Scheduler Activation
- Client/Server Implementation
  - Multithreaded Server
  - Xwindow
  - Server General Design Issues