Processes and Threads

Yücel Saygın

These slides are based on your text book and on the slides prepared by Andrew S. Tanenbaum
Sequential Computation

- **Algorithm**: collection of instructions along with their sequential order of execution
  - Algorithm is the basis for sequential execution
  - Has one entry point and instructions are executed in sequence with branches and loops until termination
- **Source program**: algorithm encoded in a programming language (Java, C, C++)
- **Computer systems have tools that can convert a source program into a binary program (object code, executable)**
- **Binary program can be executed directly by the computer**
Sequential Computation

- OS procedure calls are used inside the binary code to interact with the OS: for using I/O devices and to terminate execution.
- Programmers write their code with the assumption that it will be executed in sequential fashion.
- But OS may interleave the execution of multiple programs.
Process

- Process is a program in execution
- It includes
  - Program itself
  - Data
  - resources (such as files)
  - Execution info (process relation information kept by OS)
Processor’s Point of View

Main Memory

Program Counter
// Assembly language that does $a = b + c$
load R3, b  // Copy the value of b from memory to R3
load R4, c // Copy the value for c from memory to R4
Add R3, R4 // sum placed in R3
Store R3, a // store the sum into memory cell a
CPU

- Registers hold key variables and temporary results
- They are faster to access than the main memory
- Special Registers:
  - Program Counter contains the memory address of the next instruction to be fetched
  - Instruction Register contains the current instruction
  - Stack pointer points to the top of the current stack in memory
  - Program status word contains condition bits set by comparison instructions, and control bits
- Instructions are about loading an instruction to registers and manipulating the registers and memory locations

Registers

- Program Counter
- Stack Pointer
- Program Stat Word
- General Register 1
- General Register 2
Registers and OS

OS should be aware of the registers that keep the current state of a running program.
When OS stops running a program it should save all the registers to restore the current state of the program later on.
CPU Modes

A bit in PSW controls the mode

Kernel Mode:
- CPU can execute every instruction in its instruction set
- CPU can use every feature of the hardware
- OS runs in kernel mode

User Mode:
- Only a subset of instructions can be run
- And a subset of hardware features are accessed
- Generally all instructions involving I/O and Memory protection are disallowed in that mode

To obtain OS services, a user program must make a system call that traps into kernel mode and invokes the OS

Trap instruction switches from user mode to kernel mode and starts the operating system (system calls and errors such as division by zero causes trap instructions)
Multi programming

• A program needs to be loaded into main memory before it can be executed
• Multiple programs in RAM
• Problems:
  – How to protect programs from one another?
  – How to handle relocation?
    • We do not know which MM location the program will be loaded
    • We need to handle the addresses referenced in the program
• Solution
  – Modify all addresses when loading (X)
  – Do it on-the-fly during execution (X)
  – Use base and limit registers (++)
Multi programming

• Base register points to the start of the program text
• Limit register tells how large the combined program and data are

• When an instruction is to be fetched:
  – Hardware checks to see if the PC (program counter) is less than the limit register
  – If so adds it to the base register and sends the sum the the Main Memory.
  – Base register contents are added to every memory address reference, so program can not reference the memory below base register
  – Limit register makes it impossible to reference locations above it

• Both Protection and Relocation problems are solved with base and limit registers
Computer Hardware Review (5)

One base-limit pair and two base-limit pairs
I/O Devices

• An input device transfers data from a mechanism like keyboard, mouse, touch screen, or microphone into a CPU register
• CPU will then store the data into Main Memory
• For Output devices, CPU fetches the information from main memory to its registers and transfers this information via the bus to an output device like screen, speaker, or printer.
• Communication devices: serial and parallel ports, infrared transmitter/receivers, wireless network cards, network interface cards
Device controllers have microprocessors, they can work in parallel. They can read/write devices and memory in parallel. They can tell CPU when they are done.
Interrupts

- Once a device is started by the device driver, the application can not proceed until the operation is complete.
- The Device driver can do:
  - Polling and busy-waiting (continuously checking the flags of the device)
  - Or the device can signal the CPU when it has finished its job
Interrupts

• An interrupt request flag is added to CPU and control unit checks this flag during each fetch-execute cycle.
• All the device done flags are connected to the interrupt request flag using OR logic.
• When a “done” flag is set for any device, then the interrupt request flag is also set informing the control unit of CPU.
(a) Steps in starting an I/O device and getting interrupt
(b) How the CPU is interrupted
Processes
The Process Model

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant
Process Creation

Principal events that cause process creation

• System initialization
• Execution of a process creation system call
• User request to create a new process
Process Termination

Conditions which terminate processes
1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)
Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal
Process States (1)

- Possible process states
  - running
  - blocked
  - ready
- Transitions between states shown

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
Answer the following questions

- (True/False) At a certain time, only one process can be executing its instructions on your laptop
- A process goes into blocked state when
  - It tries to read a memory location in RAM (T/F)
  - When it tries to read a disk block (T/F)
  - When it tries a print a document (T/F)
  - When its next instruction is going to be fetched (T/F)
• **Scheduler is the lowest layer of process-structured OS**
  – handles interrupts, scheduling
• Above that layer are sequential processes
Queues Maintained by OS

• **Ready Queue**:  
  – list of processes waiting to be executed  
  – Scheduler picks a process from ready queue to execute (dispatching)

• **Blocked Queue**:  
  – List of processes waiting for I/O
Process Related OS Tables

Memory
Devices
Files
Processes

Memory Tables
I/O Tables
File Tables

Primary Process Table
Process 1
Process 2
……
Process n

Process 1 Image
……
Process n Image
Process Related Structures

• Process Location: the address of the location where we keep info about the process and process related structures
• Process control block : attributes related to a process
• Program text, data, stack, heap
• Collection of all of the above is called the process image
Process Related Structures

• **Process Location**
  
  – Process images are usually kept in the secondary storage (disk)
  
  – In order to run a process, we need to bring the process image into memory (or at least part of it)
Process Related Structures

- Process Attributes (kept in process control block)
  - Process identification
  - Processor state information
  - Process control information
Process Attributes

• **Process identification**
  - process id: identifies the process (may be an index to the primary process table)
  - parent id: id of the parent process
  - user id: id of the user responsible for the process
Process Attributes

• **Processor state information**
  – Contents of the processor registers
    • User-visible registers
    • Control and status registers
    • Stack pointers
  – When the process is interrupted, contents of the registers must be saved for restoring the process state later on
Process Attributes

- **Process control information**
  - Scheduling and state information
    - Process state (running, ready, blocked)
    - Priority
    - Scheduling related information (how much it waited, how much it used the CPU)
    - Event (the id of the event the process is waiting for)
  - Data structuring
    - queues,
    - priority levels,
    - parent-child relationships
  - Memory management
  - Resource ownership (open files etc)
  - Interprocess communication (flags, signals)
Processes in UNIX

• Two different types of processes:
  – System processes
    • run in kernel mode and execute operating system code for administrative functions like process swapping, memory management, etc
  – User processes
    • run in user mode to execute user processes
    • run in kernel mode to execute instructions from the kernel
Processes in UNIX

- There are two important processes in UNIX
  - Process 0 is a special process that is created when the system boots and its task is to do process swapping
  - Process 0 creates process 1 (the init process) which spawns the user process when the user logs on to the system

- Processes in UNIX are represented by complex data structures for managing and dispatching processes
Processes in UNIX

- **Image of a process contains:**
  - User-level context (program text, data, stack, shared memory)
  - Register context (Program counter, processor status register, stack pointer, general purpose registers)
  - System-level context
    - **Process table entry** which defines the state of the process,
    - **user area** which contains the process control information,
    - **per process region table** which does the mapping from virtual to physical addresses,
    - **kernel stack** which contains the stack frame of kernel procedures as the process executes in kernel mode.)
The Stack

- Stores information about the active subroutines of a computer program
- Its primary function is to keep track of the point to which each active subroutine should return control when it finishes executing
- A subroutine may call another subroutine and this may go on and on (nested calls) which can be easily handled through the Stack.
- The caller pushes the return address onto the stack, and the called subroutine, when it finishes, pops the return address off the call stack and transfers control to that address.
- If the pushing consumes all of the space allocated for the call stack, an error called a stack overflow occurs.
The Stack

- A stack frame contains the input parameters of the subroutine, the local variables and the return address.

<table>
<thead>
<tr>
<th>Process identification</th>
<th>Process State information</th>
<th>Process Control information</th>
</tr>
</thead>
<tbody>
<tr>
<td>User stack</td>
<td>Private user address space (Programs, data)</td>
<td>Kernel stack</td>
</tr>
<tr>
<td>Shared address space</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of stack frame](image)
The Heap

- For dynamic memory allocation (managed through system calls malloc() and free())
- Allocating, deallocating, and managing the free space is a sophisticated task and we will see more about that during memory management part of this course
Process Control in UNIX

• Kernel system call fork() is used to create a process
• When a process (parent) issues a fork() command to create a child process, the OS performs the following (in kernel mode):
  – Allocates a slot in the process table for the new process
  – Assigns a unique process ID to the child process
  – It makes a copy of the parent process image except the shared memory
  – It assigns the child process to “ready to run” state
  – It returns the ID number of the child process to the parent process, and “0” value to the child process
Process Control in UNIX

• After completing the process creation tasks, the OS can do the following
  – Stay in parent process (parent resumes in user mode at the statement just after the fork call)
  – Transfer control to the child process, and child executes from the point of the fork call
  – Transfer control to another process (both parent and child are left in ready to run state)
Threads
The Thread Model (1)

(a) Three processes each with one thread
(b) One process with three threads
The Thread Model (2)

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

- **Items shared by all threads in a process**
- **Items private to each thread**
The Thread Model (3)

Each thread has its own stack
Thread Usage (1)

A word processor with three threads
Implementing Threads in User Space

A user-level threads package
User Level Threads

- Underlying OS implements the classic processes
- A user level thread library (such as POSIX thread library) implements the threads
- OS is not aware of the threads
- Scheduling by OS is done at the process level
- Advantages:
  - Portable
  - Fast switching from one thread to another
- Disadvantages:
  - When one thread switches to kernel space and blocks (say for I/O) all the others will block
  - Threads can not be scheduled to different processors
Implementing Threads in the Kernel

A threads package managed by the kernel
Kernel Level Threads

• In Modern Operating Systems such as Solaris, Windows NT, Linux (>2.2.0) threads are implemented at kernel level
• Later version of Linux implements threads at kernel but the thread interface is still at the user level
• Threads and processes are managed separately
• Advantage:
  – Threads can be scheduled to multiple processors
  – If one thread is blocked, the others may execute
• Disadvantage:
  – Switching from one thread to the other is faster than process switch but slower than user level thread switch
Hybrid Implementations

Multiplexing user-level threads onto kernel-level threads
Hybrid Approach

• Solaris Threads (Solaris 2.x):
• Has multi-threaded architecture:
  – Process: like a unix process, includes user’s address space, stack, and process control block
  – User-level threads: Invisible to OS, implemented through threads library
  – Kernel threads: unit of scheduling at kernel level (to multiple processors if there are any)
  – Lightweight processes (LWP): mapping from user level threads to kernel threads. Each LWP handles multiple user level threads and mapped to one kernel thread
Scheduler Activations

- **Goal** – mimic functionality of kernel threads
  - gain performance of user space threads
- **Avoids unnecessary user/kernel transitions**
- **Kernel assigns virtual processors to each process**
  - lets runtime system allocate threads to processors
- **Problem:**
  Fundamental reliance on kernel (lower layer) calling procedures in user space (higher layer)
More on Windows NT processes and threads during the recitations with program fragments