Chapter 6

Process Management
OS organization

- Process and resource manager
- Memory Manager
- Device Manager
- File Manager
- Processor(s)
- Main Memory
- Devices
Process Management Tasks

- Define & implement the essential characteristics of a process and thread
  - Algorithms to define the behavior
  - Data structures to preserve the state of the execution
- Define what “things” threads in the process can reference – the *address space* (most of the “things” are memory locations)
- Manage the resources used by the processes/threads
- Tools to create/destroy/manipulate processes & threads
Process management (...ctd)

- Tools to time-multiplex the CPU – Scheduling the (Chapter 7)
- Tools to allow threads to synchronize the operation with one another (Chapters 8-9)
- Mechanisms to handle deadlock (Chapter 10)
Introduction

- Scenario
  - One process running
  - One/more process performing I/O
  - One/more process waiting on resources

- Most of the complexity stems from the need to manage multiple processes
Introduction

- Process Manager
  - CPU sharing
  - Process synchronization
  - Deadlock prevention
Process Manager Overview

Program → Process

Abstract Computing Environment

File Manager
- Device Manager
- Memory Manager
- Devices
- Memory

Deadlock
- Process Description
- Protection
- Synchronization
- Scheduler
- Resource Manager
- CPU
- Other H/W
Process components

- Program
  - defines behavior
- Data
- Resources
- Process Descriptor
  - keeps track of process during execution
### Process Descriptor

<table>
<thead>
<tr>
<th>FIELD</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal process name</td>
<td>An internal name of the process, such as an integer or table index, used in the operating system code.</td>
</tr>
<tr>
<td>State</td>
<td>The process's current state.</td>
</tr>
<tr>
<td>Owner</td>
<td>A process has an owner (identified by the owner's internal identification such as the login name). The descriptor contains a field for storing the owner identification.</td>
</tr>
<tr>
<td>Parent process descriptor</td>
<td>A pointer to the process descriptor of this process's parent.</td>
</tr>
<tr>
<td>List of child process descriptors</td>
<td>A pointer to a list of the child processes of this process.</td>
</tr>
<tr>
<td>List of reusable resources</td>
<td>A pointer to a list of reusable resource types held by the process. Each resource type will be a descriptor of the number of units of the resource.</td>
</tr>
<tr>
<td>List of consumable resources</td>
<td>Similar to the reusable resource list (see Section 6.3.2).</td>
</tr>
<tr>
<td>List of file descriptors</td>
<td>A special case of the reusable resource list.</td>
</tr>
<tr>
<td>Message queue</td>
<td>A special case of the consumable resource list.</td>
</tr>
<tr>
<td>Protection domain</td>
<td>A description of the access rights currently held by the process (see Chapter 14).</td>
</tr>
<tr>
<td>CPU status register content</td>
<td>A copy of each of the CPU status registers at the last time the process exited the running state.</td>
</tr>
<tr>
<td>CPU general register content</td>
<td>A copy of each of the CPU general registers at the last time the process exited the running state.</td>
</tr>
</tbody>
</table>
Process Address Space

- Defines all aspects of process computation
  - Program
  - Variables
  - ...
- Address space is generated/defined by translation
Creating an executable program

Source Modules
Translator
Relocatable Object Modules
Link Editor
Absolute Program
Loader
Executable Program

Separate objects each relative to 0
One large program 0 - X
Y - (X+Y)
Maps relative address space to physical memory location
Relocates modules one behind other
- Relocates addresses of all but first
- Resolves external reference to library calls and external modules

Generates separate object code modules
Basic Memory Hierarchy

Access Speed

Fastest

Slowest

Cache memory

Primary Memory, $M_p$

Secondary Memory, $M_s$
Basic Memory Hierarchy...

- At any point in the same program, element can be in:
  - Secondary memory $M_S$
  - Primary memory $M_P$
  - Registers $M_R$

- **Consistency is a Problem**
  - $M_S \neq M_P \neq M_R$ (code vs data)
  - When does one make them consistent?
  - How?
Consistency Problem

- Scheduler switching out processes – Context Switch
- Is Instruction a Problem ???
  - NO
  - Instructions are never modified
  - Separate Instruction and Data space
  - Therefore, $M_{R_j} = M_{P_j} = M_{S_j}$

How can an instruction be in a register?
Consistency Problem...

- Is Data a Problem ???
  - YES
  - Variable temporarily stored in register has value added to it
  - Therefore, \( M_{Rj} \neq M_{pj} \)

- On context switch, all registers are saved
  - Therefore, current state is saved
Sample Scenario...

- Suppose ‘MOV X Y’ instruction is executed
  - \( M_{Py} \neq M_{Sy} \)

- On context switch, is all of a process’ memory flushed to \( M_S \)?
  - No, only on page swap

- Hence, \( env_{process} = (M_R + M_S) + (\ldots) \)

- Note:
  - Flushing of memory frees it up for incoming process
    - => Page Swap
Process States

- Focus on Resource Management & Process Management

- Recall also that part of the process environment is its state

State Transition Diagram
Process States...

1. When process enters ‘Ready’ state, it must compete for CPU. Memory has already been allocated

2. Process has CPU

3. Process requests resource that is \textit{immediately} available $\rightarrow$ NO blocking

4. Process requests resource that is \textit{NOT} yet available

5. Resource allocated, memory re-allocated?

State Transition Diagram
Resources & Resource Manager

- 2 types of Resources
  - Reusable (Memory)
  - Consumable (Input/Time slice)

![Diagram of resource management process]

- Process requesting resource unit(s)
  - Get it, or
  - Block => Stay in Queue
Resource Descriptor

- Each Resource R has a Resource Descriptor associated with it (similar to the process)
  => there is a “Status” for that Resource, and
  => a Resource Manager to manage it

<table>
<thead>
<tr>
<th>FIELD</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal resource name</td>
<td>An internal name for the resource used by the operating system code. /dev/...</td>
</tr>
<tr>
<td>Total units</td>
<td>The number of units of this resource type configured into the system. 6</td>
</tr>
<tr>
<td>* Available units</td>
<td>The number of units currently available. 3</td>
</tr>
<tr>
<td>List of available units</td>
<td>The set of available units of this resource type that are available for use by processes. A, B, C</td>
</tr>
<tr>
<td>List of blocked processes</td>
<td>The list of processes that have a pending request for units of this resource type. Only if * = 0</td>
</tr>
</tbody>
</table>
- Conceptually, this is the way in which we would like to view it
- Root controls all processes i.e. Parent
Creating Processes

- Parent Process needs ability to
  - Block child
  - Activate child
  - Destroy child
  - Allocate resources to child

- True for User processes spawning child
- True for OS spawning `init, getty, etc.`
- Process hierarchy a natural,
  
  if `fork/exec` commands exist
Factoring in additional Control Complexities

- **Recall:**
  - A parent process can **suspend** a child process

- Therefore, if a child is in **run** state and goes to ready (time slice up), and the parent runs and decides to suspend the child, then how do we reflect this in the process state diagram ???

- **We need 2 more states**
  - Ready suspended
  - Blocked suspended
Process State diagram reflecting Control

- Not Blocked
- Suspended
- No memory

- Blocked
- Suspended
- No memory

- Not blocked
- Not suspended
- Has memory

- Blocked
- Not suspended
- No memory
Why can a process NOT go from ‘Ready Active’ to ‘Blocked Active’ or ‘Blocked Suspended’?
Process Management under Linux

Mir Farooq Ali
Processes in Linux

- Also called \textit{tasks}

- Task table or process table defined in src/linux/include/sched.h

  \begin{verbatim}
  extern struct task_struct
    *pidhash[PIDHASH_SZ];
  \end{verbatim}

- Can also be accessed as a doubly-linked list $p->\text{next}\_\text{task}$ \textbf{and} $p->\text{prev}\_\text{task}$
Process or task descriptor

- Called task_struct
- Present in src/include/linux/sched.h
- Contains various fields to indicate:
  - state
  - priority
  - pointers to parent, children, other tasks in pid list
  - tty
  - memory location
  - file descriptors
  - ...

Fall 1999 : CS 3204 - Arthur
Process States

- Linux identifies following states
  1. TASK_RUNNING
  2. TASK_INTERRUPTIBLE
  3. TASK_UNINTERRUPTIBLE
  4. TASK_ZOMBIE
  5. TASK_STOPPED
  6. TASK_EXCLUSIVE
Process Creation

- Remember in traditional UNIX, we use `fork()` and then typically `exec()`.
- `fork()` duplicates resources owned by parent for child process and copies them to new address space.
- This method is slow and inefficient, since `exec()` wipes out address space anyway.
Process creation in Linux

- Copy On Write technique
- Lightweight processes
- vfork()
Copy-on-write

- Child pages are pointers to parent pages
- If child makes a change to a page, a new copy is made for the child
- This way, you avoid making separate copies of pages unnecessarily
Lightweight processes

- Allow parent and child processes to share many kernel data structures
- created in Linux by function called __clone()
- uses non-standard clone() system call
vfork()

- Creates a process that shares memory address of parent
- Parent is blocked until child exits or executes a new program by doing exec()
User view of processes

- Can use ps command with various options, for example,
  - ps –aux
  - ps –ef
/proc file system

- process information pseudo file system
- Do `man proc` to get more info
- `/proc` directory contains
  - Numerical subdirectory for each running process
  - A number of other files containing kernel table information
 mús& Fe] lecluce

■ cpuinfo – contains CPU specs
■ uptime – time in secs since machine was last rebooted and idle time since then
■ version – kernel version
■ loadavg – Load average of machine over the past 1, 5 and 15 minutes
■ ...

Fall 1999 : CS 3204 - Arthur
Process directories

- One subdirectory for each running process
- Files include
  - cmdline
  - cwd
  - environ
  - exe
  - fdm
  - map
  - mem
  - root
References

- Linux Kernel 2.4 internals, Tigran Aivazian [http://www.tldp.org/LDP/lki/](http://www.tldp.org/LDP/lki/)
- Modern Operating Systems, 2nd Ed., A. Tanenbaum
- Understanding the Linux Kernel, D. Bovet, and M. Cesati