SE 350 — Operating Systems

Kevin James

Winter 2015

Contents

1

“‘ Availability Concerned with protecting the system against interruption

Confidentiality Assuring users cant read data for which access in unauterized

Data integroity protection of data from unauthorized modificaton

Authenticity Concerned with the proper verification of the identity of users and the validity of messages and/or data

Fairness Give equall and fair access to resources of OS, so apps get CPU access when needed.

Differential Responsiveneess Discriminate among different job classes, often called ‘Quality of Service‘. This is a necessity for realtime OS.

Efficieny Maximize throughput, minimize response time, and accommodate as many users as possible

Problems with OSs Usually late on delivery Latent bugs (not quite banaga products but already close) Unexpectedly slow performance Security vulnerabilities

A good, hierarchical structure may help with this.

Structure View the system as a series of levels: each level performs a related subset of functions. Each level relies on lower levels to perform more primitive functoins. This decomposes a problem into a number of more manageable subproblems.

Modern OS MicroKernel Archs Assigns only a few essential functions to the kernel (address spaces, IPC, basic scheduling). All other elements are in user space. QNX is a popular example of this architecture.

Multithreading Process is divided into threads that can run concurently. Thread: dispatchable unit of work, executes sequentially and is interruptable. Process: collection of one or more threads.

Symmetric MultiProcessing There are multiple processors, which share the same main memory and IO facilities. All processors, then, can perform the same functions.

Distributed OS Provides the illusion of a single main memory space and single secondary memory space

Assymetric multiprocessing Moderate success of prpocessor: “big-little architectures”

Object Oriented design Add support for modularity ”’

2 Process Description and Control

Use multiple process to maximize CPU utilization while providing reasonable resaponse time. Allocate resoures to processes and suppport interprocess communication and user creation of processes. With a **uniprocesssor**, we interleave execution of processes. For a **multiprocessor**, interleaving and parallel execution of processes both.

A **process** is a program in execution, OR an instance of a program running on the computer, OR the entity that can be assigned to and executed by the CPU. It consists of three componenets: an executable program, associated data needed by the program, execution context of the program (all information the OS needs to manage the process).

- Identifier: PID
- State: running state
- Priority: relative to other processes
- Memory Pointers: pointers and shared memory blocks
- Context data: registers, PSW, PC
- IO Status Info: outstanding IO requests, used IO devices
- Accounting Info: amount of CPU time, time limits, threads

A **process control block** is a data structure that contains process elements; it is created and managed by OS.

A process at any point may be either running or not-running. Dispatching moves it to the running state and pausing moves it to the not-running state. Note that this is in addition to a process *not existing.*

2.1 Two-State Process Model

(process may be in one of two states...)

Simple queueing mechanism is inefficients because some processes are not running but ready or nt running and blocked, etc. With a single queue, we must scan the full list for a not-running, ready item which has been there the longest. With multiple queues, we can pick the correct queue and then choose a process "round robin" style.

Five state model:

- Running: a currently executing process
- Ready: a process ready to be picked up
- Blocked / Waiting: blocked because it is waiting for something
- New: a new process
- Exit: a halted / aborted process

2.2 Suspended Processes

(problem...)

2.3 Process Control Blocks

Each process will have a PCB that stores info about the process state:

- Process State
- ID
- Priority
- PC
- CPU registers
- SP

Process switching follows the following process: select next process with scheduler, invoke context switch to new process. The context switch is slightly more complicated:

1. Save context of current process (PC and registers)
2. Change process state to ready
3. Update current_process to point to new process
4. Set new process state to executing
5. Restore context of current_process
6. Execute current_process

Note that a complete process switch involves switching back to the original process, ie. flipping between the two processes. In this case, we simply execute the context switch again.

We may have **non-process kernels** which:

- execute the kernel outside of any process
- have their own memory and call stack
- the conecpt of processes only applies to user programs
- OS is exuected as a separate entity running in privileged mode
- all OS calls are blocking

We can also run the kernel within user processes. The OS, then, is primarily a collection of routines: system software within the context of a user process. This process executes in privileged mode when executing OS code and performs a context switch when entering a system call (though it continues with the same process).

3 Monitors

Monitors are software modules. They form a mutex lock with controlled access such that only one process can use a monitor at one time, and only to access local or shared variables. Monitors use condition varibales for signalling and lose unused signals (except semaphores).