**CS 153**

**Design of Operating Systems**

**Ch. 3 Multiprogramming**

*multiprogramming* – an OS that can run several programs at a time

Allows for overlapped use of resources.

When two users use a program at the same time, there is one program running, but two *processes* running.

Programs can execute another by using a system call whose handler is called the *loader*. The invoking program stops and never resumes under the process.

*process* – a sequence of program executions each of which results from its predecessor via an invocation of execute. (aka: *task, heavy-weight process*)

*thread* – stream of activity to which CPUs are allocated (aka: *light-weight processes*)

Processes are the fundamental units of dynamic binding to system resources (except CPUs).

**3.1 Process Anatomy**

A process is always running an executable, which contains instructions for a collection of procedures with initial values for global variables.

An execution image consists of the state of the memory segments, status of open-files, register settings, etc.

**code segment**

contains the machine-language instructions for a set of procedures that includes those that the threads are running. Read only.

**data segments**

contains the heap, space for global variables, I/O buffers, and transient portion of the process’s descriptor.

organization:

* Initialized data
* Uninitialized data segment
* Heap
* Free area/stack growth
* argv[]
* envp[]

**stacks**

stores activation record for procedure invocation

AR contains local variables, saved values of registers, passed parameters, return values, etc.

**descriptors**

includes tables that hold the identities of its data segment and its open-files plus information about its thread(s).

resident part: contains priority information and enough information to make the process resident in main memory

transient part: contains all other information needed to resume the process’s threads and keep track of its status and its connections to resources

Processes are not allowed to write directly to their descriptors.

**3.2 Loading**

When a program executes another, the old process descriptor remains. The stack and private data segments are usually replaced by fresh ones. No new process is created.

An exec invocation returns iff there is an error.

Code segments are shared among processes. They are created by exec only when necessary.

**3.3 Process Creation and Termination**

1. Kernel is loaded
2. Root processes of process tree created
3. Root processes creates all other initial processes (daemons)

In UNIX, two processes can communicate via any pipe established by a common ancestor.

*child process* – a clone of the parent process that inherits the parent’s code segment and register settings. It also gets a duplicate copy of the parent’s data segments. stacks, and descriptors.

Termination

All open-files closed. Space is deallocated, descriptor returned to free-space or free-descriptor pool.

In UNIX, two system calls for handling termination

exit(status) – terminates the calling process and communicates the value of the integer parameter status to the caller’s parent

pid = wait(&status) – caller waits until one of its children exits. PID of child returned; status is reported by the exiting child in status

Use child processes and wait to retain control of the parent process.

fork is easy to implement when there is sufficient memory-mapping support, where each process has its own address space. Difficult when a process has to deal with absolute addresses.

**3.4 Process Migration**

Processes can migrate from one computer to another.

Migrated processes execute remotely as though all kernel invocations were executed via RPCs on the migrator’s home machine.

**3.5 Daemons**

There are other processes called daemons for the OS to handle chores. (ex: printer control)

UNIX:

init – the most important one and ancestor of all user processes and most system processes.

inetd – manages other daemons

**Linux bootup**

boot manager invoked on the MBR

contains copies of OS kernels to bootstrap

GRUB passes arguments to the bootstrapped kernel

*init* – contains the name of a normal executable that the kernel runs to create its first daemon