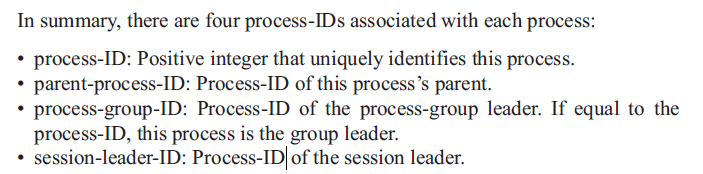
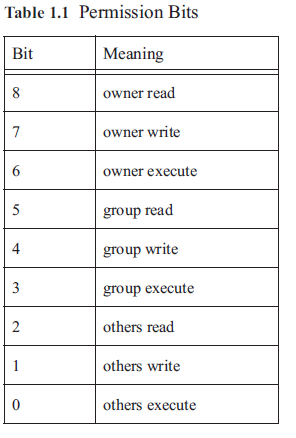
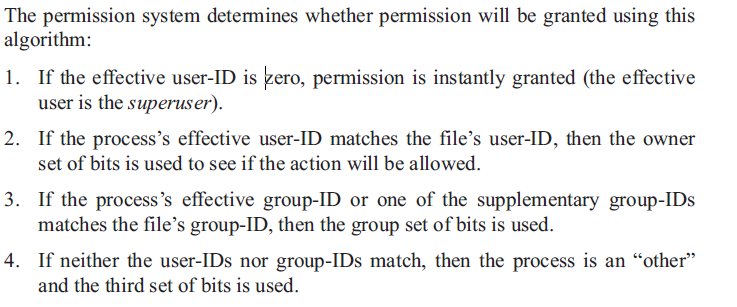
Midterm Review:

Chapter 1: Fundamental Concepts

1. Files: Regular files, directories, symbolic links, special files, named pipes and sockets;
2. Regular Files: contain bytes of data, organized into linear array, read and write start at file offset, stored on disk; can not insert or delete from middle; multiple processes may work on file same time, system-wide flags are used to test and set; i-number is index into an array of i-nodes, contains type of file, number of links, owner id and group id, three sets of permission, size in bytes, time of last access, last modification , status change, pointers to disk blocks containinf file’s contents;
3. Directories and Symbolic Links: allow names to be used, almost always used to access a file. 2-column table: name -> i-number; directory itself occupy an i-node and have data. Hierarchical structure: memo/july/smith; relative path starts with current dir, system keeps track of i-number of current dir; absolute path begins with / and starts with root dir; kernel reserve inumber for root, when file system is first constructed. Program can not write a dir as two column is directly used by kernel and might mess up the whole system. Program manipulates a directory by using system calls, legal writing actions are to add or remove a link.; two or more links may refer to same file. No ambiguous when accessing, i-node keeps link count for removing a link ; multiple links to a file using i-number only work if links in same file system, symbolic links put path of file to be linked to in the data part of an actual file, no read or write but use system calls.
4. Special file typically points to device; block and character; big block for cache, character for direct access and high-performance. Special file has i-node, but no data instead, a device number.
5. Program, processes, and threads:

* Program, collection of instruction and data that is kept in a regular file on disk, marked executable in i-node; contents obey rules established by the kernel; program into text-file -> object file with machine-language translation; linker is used to bind object file with libraries
* Process, is created to run a program; contains three segments: instruction seg, user data seg, system data seg (include current dic, open file des, accumulated CPU time);
* Threads, are tracked by kernel: separate flow of control through instructions, start with one thread, unless execute special system call to create another;
* There is no functional relationship between processes initialized from the same program ( process can not detect shared instruction seg); strong functional replationship between threads in the same process
* Process has inheritance, threads are equal ( all threads have equal access to all data and resources, not copy)

1. Signal: sent by kernel, process, another process or user; segmentation-violation signal is generated by kernel; ‘abort’ is sent by process to terminate with a core dump; termination signal, sent by another process; ctrl-c sent by user
2. Process-ID, process groups, session: process id are guaranteed to be unique; every process but one has a parent; process record its parent id; parent id set to some fix number if parent process terminated; initialization process has all orphan processes. Related processes are organized into process group; process groups are further organized into sessions; session has one session-leader, process group has one process-group leader; Unix shell creates one session per login, form a separate process group for the processes in a pipeline; kernel allows any number of sessions per login as long as hierarchy is maintained; session have controlling terminal opened by session leader, become controlling process; job control: one process per group , move process groups back and forth, background process suspended until moved to foreground; closing a terminal will terminate all user’s processes in that group unless otherwise specified. 
3. Permissions: user-ID is a positive integer associated with a user’s login name in password file; login command makes this ID the user-ID of the first process created, and descended from the shell inherit this user-ID; users organized into group, with group-ID, user’s login group-ID is made the group ID of login shell; real-ID can be different from effective-ID; effective -ID used to determine permission, real id to determine identity; i-node keeps owner user-id and owner group-id; three sets of three permission bits, each has one bit for read, one for write and one for execute; 1 is granted; 0 is not; permission stored as octal number

* Permission system algorithm: 

1. Inter-process communication:

* Shared file offsets: one process could position the file offset to some fictitious location in a file, second process figured out where it points;
* Signals: one process send to another
* Process tracing: parent control the execution of its child
* Files: common form of inter-process communication, eg. One process vi to write, another process python to run;
* Semaphore: is a counter that prevents two or more processes from accessing the same resource at the same time
* File lock: a special semaphore, prevents two or more processes accessing the same segment of a file; weak form: advisory locking, stronger form: mandatory locking;
* Message: small amount of data sent to message queue
* Shared memory: provided fastest inter-process communication
* Sockets: networking inter-process communication, using a group of system calls called sockets; processes in communication doesn’t have to be on the same machine;

1. Versions of UNIX:
2. Using the system calls:

* C and C++ bindings, small function that executes some special code that transfers control from the user process to the kernel and then returns the results; AVOID EXCESSIVE SYSTEM CALLS; every system call is defined in a header file;
* Guideline:

1. Include necessary headers
2. Include error handling
3. Don’t use cast unless absolutely necessary
4. If more than one thread, make sure function to call is thread sage
5. Use standard interfaces rather than particular system
6. Function synopses: test for error handling
7. Error Handling: system call returns a value that can’t be mistaken for valid data, typically -1; 80% system calls, indicates reason through errno (errno.h), used as an integer; don’t declare errno but use the definition from header;
8. Error checking macros: use utility function to avoid excessive repeating codes; other 20 % that doesn’t set the errno will return the error code; macros give: easy and readable error checking, an automatic jump to cleanupcode, complete error info; macros have weird syntax
9. Date and time:

* Calendar time: used for access , modification, and status-change times of files; four forms: arithmetic type: time\_t, count of numbere of seconds since epoch, set to midnight, January 1, 1970; structure type: struct timeval, hold time in seconds and microseconds; structure type tm, hold time in year, month, day, hour, minute, second, and gew other things; string
* Execution Time: measure timing intervals and process-execution times; clock\_t, arithmetic type, time interval in units of CLOCKS\_PER\_SEC or clock ticks; struct timeval holds an interval in seconds and microseconds; struct timespec holds an interval in sec and nanosec;