Unix 101

- Best starting point: read the paper “The Unix Timesharing System” by Ritchie and Thompson (1974)

5. Processes and Images

An *image* is a computer execution environment. It includes a core image, general register values, status of open files, current directory, and the like. An image is the current state of a pseudo computer.

A *process* is the execution of an image. While the processor is executing on behalf of a process, the image must reside in core; during the execution of other processes it remains in core unless the appearance of an active, higher-priority process forces it to be swapped out to the fixed-head disk.

Unix 101

For most users, communication with UNIX is carried on with the aid of a program called the Shell. The Shell is a command line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs. In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

command arg_1 arg_2 $\cdots$ arg_n

The Shell splits up the command name and the arguments into separate strings. Then a file with name *command* is sought; *command* may be a path name including the "/" character to specify any file in the system. If *command* is found, it is brought into core and executed. The arguments collected by the Shell are accessible to the command. When the command is finished, the Shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file *command* cannot be found, the Shell prefixes the string */bin/* to command and attempts again to find the file. Directory */bin* contains all the commands intended to be generally used.

Time in Unix

A Linux system actually has two clocks: One is the battery powered "Real Time Clock" (also known as the "RTC", "CMOS clock", or "Hardware clock") which keeps track of time when the system is turned off but is not used when the system is running. The other is the "system clock" (sometimes called the "kernel clock" or "software clock") which is a software counter based on the timer interrupt. It does not exist when the system is not running, so it has to be initialized from the RTC (or some other time source) at boot time. References to "the clock" in the `ntpd` documentation refer to the system clock, not the RTC.

The two clocks will drift at different rates, so they will gradually drift apart from each other, and also away from the "real" time. The simplest way to keep them on time is to measure their drift rates and apply correction factors in software. Since the RTC is only used when the system is not running, the correction factor is applied when the clock is read at boot time, using `clock(8)` or `hwclock(8)`. The system clock is corrected by adjusting the rate at which the system time is advanced with each timer interrupt, using `adjtimex(8)`.

Time in Unix

Time keeping and use of clocks is a fundamental aspect of operating system implementation, and thus of Linux. Clock related services in operating systems fall into a number of different categories:

- time keeping
- clock synchronization
- time-of-day representation
- next event interrupt scheduling
- process and in-kernel timers
- process accounting
- process profiling

Time in Unix

The Linux "system clock" actually just counts the number of seconds past Jan 1, 1970, and is always in UTC (or GMT, which is technically different but close enough that casual users tend to use both terms interchangeably). UTC does not change as DST comes and goes— what changes is the *conversion* between UTC and local time. The translation to local time is done by library functions that are linked into the application programs.

This has two consequences: First, any application that needs to know the local time also needs to know what time zone you're in, and whether DST is in effect or not (see the next section for more on time zones). Second, there is no provision in the kernel to change either the system clock or the RTC as DST comes and goes, because UTC doesn't change. Therefore, machines that only run Linux should have the RTC set to UTC, not local time.

Time in Unix

```
//int gettimeofday(struct timeval *tv, struct timezone *tz);
// struct timeval {
//              time_t      tv_sec;     /* seconds */
//              suseconds_t tv_usec;    /* microseconds */
//  };
double timestamp()
{
 double rc = DOUBLE_NOERROR;
  struct timeval tv;
  if (gettimeofday(&tv, NULL) < 0) {
     printf("utils:timestampt: gettimeofday failed, errno: %d \n",errno);
     rc=DOUBLE_ERROR;
  } else {
    rc = (double)tv.tv_sec + ((double)tv.tv_usec / 1000000);
  }
  return rc;
}
```

Time in Unix

```
// int clock_gettime(clockid_t clk_id, struct timespec *tp);
 struct timespec {
        time_t   tv_sec;        /* seconds */
        long     tv_nsec;       /* nanoseconds */
    };

//clockid_t
 CLOCK_REALTIME
        System-wide clock that measures real (i.e., wall-clock) time.  Setting  this  clock  requires  appropriate
        privileges.   This clock is affected by discontinuous jumps in the system time (e.g., if the system admin-
        istrator manually changes the clock), and by the incremental adjustments performed by adjtime(3) and NTP.
   CLOCK_MONOTONIC
        Clock  that cannot be set and represents monotonic time since some unspecified starting point.  This clock
        is not affected by discontinuous jumps in the system time (e.g.,  if  the  system  administrator  manually
        changes the clock), but is affected by the incremental adjustments performed by adjtime(3) and NTP.
     CLOCK_MONOTONIC_RAW (since Linux 2.6.28; Linux-specific)
        Similar to CLOCK_MONOTONIC, but provides access to a raw hardware-based time that is not  subject  to  N
        adjustments or the incremental adjustments performed by adjtime(3).
```

Time in Unix

```
//returns an accurate wallclock time
double getCurTime(struct timespec *ts)
{
  double timestamp = -1.0;
  int rc = NOERROR;

  //likely to use CLOCK_REALTIME
  rc = clock_gettime(CLOCK_REALTIME, ts);

  if (rc==NOERROR) {
      timestamp = ( (double)ts->tv_sec +  (double) (((double)ts->tv_nsec)/1000000000) );
  } else {
    printf("getCurTime:  HARD error on clock_gettime,  errno:%d \n", errno);
    perror("getCurTimeD:  HARD error on clock_gettime\n");
  }
  return(timestamp);

}
```

Time in Unix

```
//timestamp -   does not need wallclock time.  Just an accurate system time counter

double getTS ()
{
double myTS=-1;
struct timespec curTime;
int rc = 0;

 //Use clock_gettime
 rc =clock_gettime(CLOCK_MONOTONIC_RAW, &curTime);
 if (rc==0)
   myTS = ((double)curTime.tv_sec + ((double)curTime.tv_nsec)/1000000000);
 else{
   perror("getTS: Error on clock_gettime \n");
 }

  return myTS;
}
```

With that out of the way, let's take a look at how Linux keeps time. It starts when the system boots up, when Linux gets the current time from the RTC (Real Time Clock). This is a hardware clock that is powered by a battery so it continues to run even when the machine is powered off. In most cases it is not particularly accurate, since it is driven from a cheap crystal oscillator whose frequency can vary depending on temperature and other factors.[2] The boot time retrieved from the RTC is stored in memory in the kernel, and is used as an offset later by code that derives wall-clock time from the combination of boot time and the tick count kept by the TSC.

Time in Unix

//When doing accurate time measurements, the overhead of system calls adds to the inaccuracy

Double TS1 = -1.0;
Double TS2 = -1.0;
Double Overhead = -1.0;

TS1=getTS();
(void) getTS();
TS2=getTS();
Overhead = TS2 – TS1;
//Turns out we can directly
Read the TSC with
An assembly instruction.
This is the lowest overhead
And most accurate method
To obtain a TS!!

Both of these problems are solved in more recent CPUs: a *constant* TSC keeps all TSC's synchronized across all cores in a system, and an *invariant* (or *nonstop*) TSC keeps the TSC running at a fixed rate regardless of changes in CPU frequency. To check whether your CPU supports one or both, execute the following and examine the values output in flags:

```
$ cat /proc/cpuinfo | grep -i tsc
flags : ... tsc  rdtscp constant_tsc nonstop_tsc ...
```

Uses of time

- Wall clock time
- Accurate timings (accurate timestamps)

Unix Processes

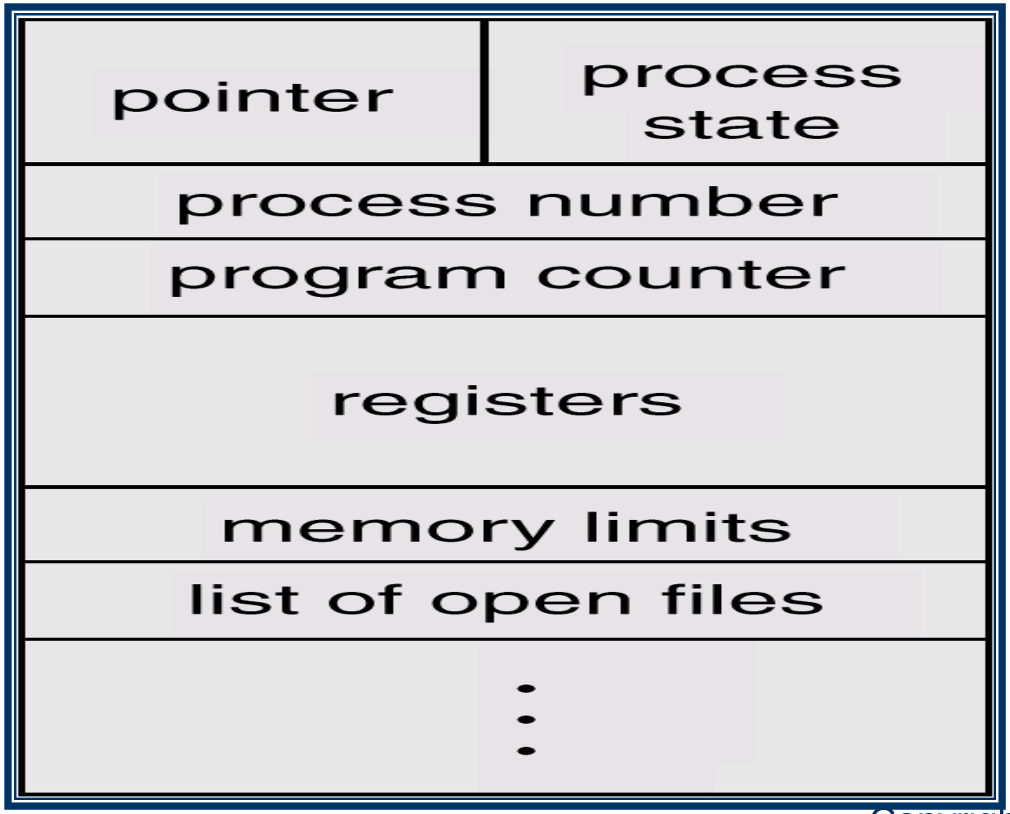
- A process is a program in execution.
  - The unit of work in a time-sharing system.
- A process has:
  - A text section: code
  - A data section: global data
  - A stack section: temp data
  - State: defined by the current activity
    - New, running, waiting, ready, terminated

Unix Processes

Information associated with each process.

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block (PCB)



Process Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.

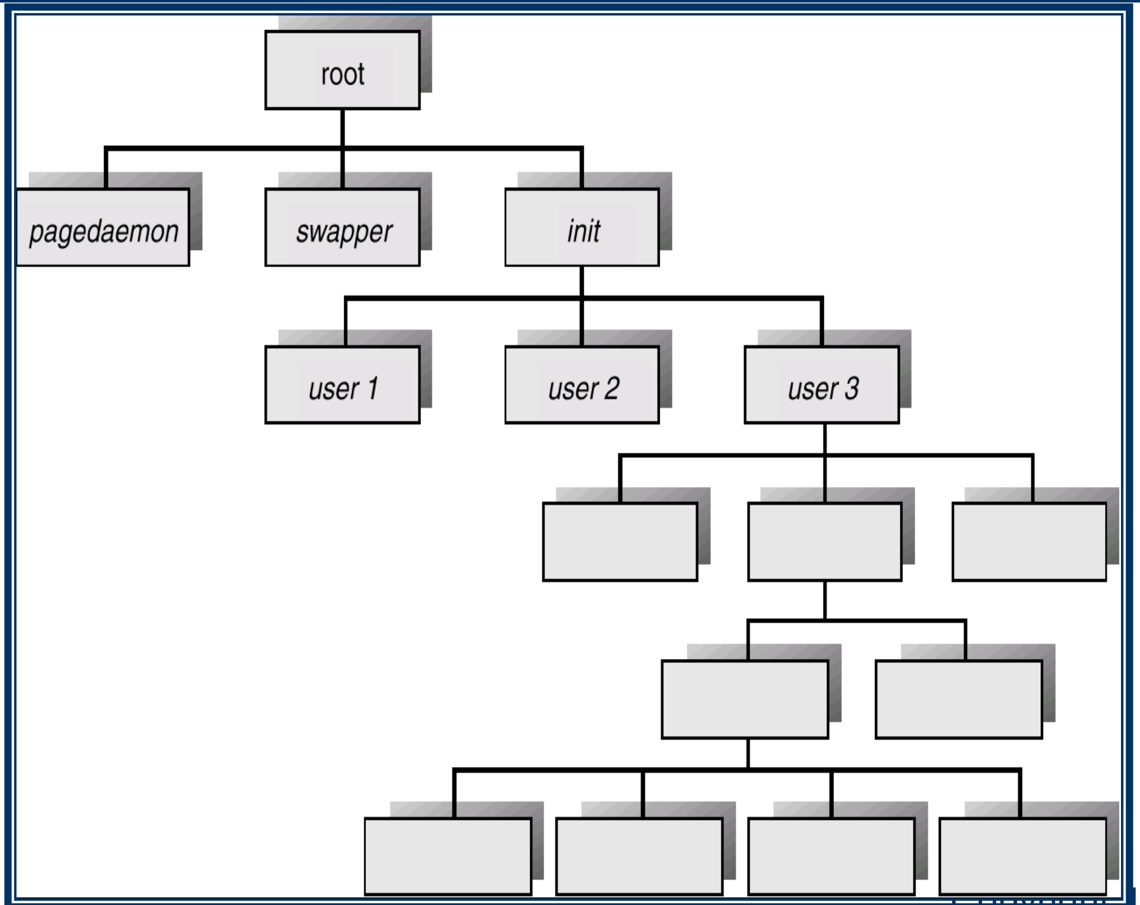
Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes.
- Resource sharing
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.
- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

Process Creation

- Address space
  - Child duplicate of parent.
  - Child has a program loaded into it.
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program.

Unix Process Tree



Unix Processes

```
blade33.cs.clemson.edu[45] ps -aux
USER       PID %CPU %MEM   SZ  RSS TT       S    START  TIME COMMAND
jeh       7271 96.5 14.21796816968 ?        R 16:33:04 20:57 ./ms_research.sparc -nice 19
root      7276  0.2  2.5 4800 2896 ?        S 16:50:03  0:00 /usr/sbin/sshd
root      7307  0.2  0.7 1104  760 pts/4    O 16:54:06  0:00 ps -aux
jmarty    7279  0.1  2.2 3336 2592 pts/4    S 16:50:06  0:00 -tcsh
root       167  0.1  1.8 4168 2152 ?        S   Feb 12  0:26 /usr/lib/autofs/automountd
root         3  0.1  0.0    0    0 ?        S   Feb 12  1:31 fsflush
root       200  0.0  1.9 3328 2280 ?        S   Feb 12  0:03 /usr/sbin/nscd
root         0  0.0  0.0    0    0 ?        T   Feb 12  0:16 sched
root         1  0.0  0.2 1224  176 ?        S   Feb 12  0:00 /etc/init -
root         2  0.0  0.0    0    0 ?        S   Feb 12  0:00 pageout
root        48  0.0  0.6 2280  696 ?        S   Feb 12  0:00 /usr/lib/sysevent/syseventd
root        55  0.0  0.9 2824  992 ?        S   Feb 12  0:02 /usr/lib/picl/picld
```

Fork

NAME

fork, fork1 - create a new process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

pid_t fork1(void);
```

RETURN VALUES

Upon successful completion, fork() and fork1() return 0 to the child process and return the process ID of the child process to the parent process. Otherwise, (pid_t)-1 is returned to the parent process, no child process is created, and errno is set to indicate the error.

ERRORS

The fork() function will fail if:

EAGAIN

The system-imposed limit on the total number of processes under execution by a single user has been exceeded; or the total amount of system memory available is temporarily insufficient to duplicate this process.

ENOMEM

There is not enough swap space.

Fork

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
  pid_t child_pid;

  printf ("the main program process id is %d\n", (int) getpid ());

  child_pid = fork ();
  if (child_pid != 0) {
   //Parents code here
    printf ("this is the parent process, with id %d\n", (int) getpid ());
    printf ("the child's process id is %d\n", (int) child_pid);
    exit(0)
  }
  else  {
   //childs code here
    printf ("this is the child process, with id %d\n", (int) getpid ());
  }

  return 0;
}
```

Zombie Process

- A process that has terminated but has not been cleaned up yet.
  - A process is responsible for cleaning up its children

NAME
    waitpid - wait for child process to change state

SYNOPSIS
    #include <sys/types.h>
    #include <sys/wait.h>

    pid_t waitpid(pid_t pid, int *stat_loc, int options);

DESCRIPTION
    The waitpid() function will suspend execution of the calling thread until status information for one of its terminated child processes is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

Zombie Process

pid_t waitpid(pid_t pid, int *stat_loc, int options);

- A pid of -1 is a wildcard that says take any zombie.
- The pid of the child process that was terminated is returned.
  - A ‘-1’ is returned on error (check errno)
  - A ‘0’ is returned if there are no zombies
- Stat_loc returns the termination status of the child.
  - Can use a NULL if not interested
- The options allows further options such as:
  - WNOHANG which tells the kernel not to block if there are no terminated children

Waitpid System Call

```
extern int currentNumberOfClients;
void
sig_chld(int signo)
{
pid_t   pid;
int stat;

  printf("sigchldwaitpid: Entered ....\n");
  while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0) {
    if (currentNumberOfClients > 0)
      currentNumberOfClients--;
    printf("sigchldwaitpid:  %d terminated, currentNumberOfClients %d\n",
(int) pid,currentNumberOfClients);

  }
 return;
}
```

Multiplexing: I/O over multiple channels

- A program can wait for events that occur on any number of sockets or file descriptors
  - Sockets and file descriptors are added to a list (called a vector)
- Select() function suspends a program until one of the descriptors in the vector has an event
  - Can specify if interested in different types of events:
    - readDescs: Descriptors in this vector are checked for immediate input data availability
    - writeDescs: Descriptors in this vector are checked to see if they can be written to
    - exceptionDescs: Descriptors in this vector are checked for pending exceptions

Multiplexing: I/O over multiple channels

- int select(int maxDescPlus1, fd_set *readDescs, fd_set*writeDescs, fd_set *exceptionDescs, struct timeval *timeout)
  - maxDescPlus1: the max descriptor value
  - The three fd_set * are the three vectors
  - Timeout: If not null, this is how long the select waits.
  - Returns 0 on timeout, else number of descriptors that have events ready. A return of <0 indicates an error (check the global errno to get the specific error)

Select: setup for events over a UDP and TCP socket

```
fd_set sock_set;

int max_d = (sock_tcp > sock_udp) ? sock_tcp : sock_udp;
FD_ZERO(&sock_set);
FD_SET(sock_tcp,&sock_set);
FD_SET(sock_udp,&sock_set);

if (select(max_d + 1,&sock_set,NULL,NULL,0) == 0) {
    continue;
} else {
        if (FD_ISSET(sock_udp,&sock_set)) {
            handle_UDP(sock_udp, dropRate);
        }
        if (FD_ISSET(sock_tcp, &sock_set)) {
        }
```

malloc/free System Calls

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
//  char *strcat(char *s1, const char *s2);
//  appends contents of string s2 to s1.  Returns s1
int main() {
   char* str1;
   char* str2;
   char* str3;
   while (1)
   {
     str1 = (char*) malloc ( sizeof(char) * 10000000 );
     if (str1 == NULL) {
       printf("Malloc 1 error.....\n");
       break;
     }
     str2 = (char*) malloc ( sizeof(char) * 10000000 );
     if (str2 == NULL) {
       printf("Malloc 2 error.....\n");
       break;
     }
     str3 = (char*) malloc ( sizeof(char) * 10000000 );
     if (str3 == NULL) {
       printf("Malloc 3 error.....\n");
       break;
     }
   strcpy(str1, "abc");
   strcpy(str2, "def");
   str3 = strcat(str1, str2);
   printf ("str1: %s, str2: %s, str3: %s \n", str1, str2, str3 );
   free(str1);
   free(str2);
   free(str3);
  }
}
```

Memory leak!!!

The gdb debugger

http://www.cs.princeton.edu/~benjasik/gdb/gdbtut.html

GDB Tutorial

A debugger is a program that runs other programs, allowing the user to exercise control over these programs, and to examine variables when problems arise. The most popular debugger for UNIX systems is GDB, the GNU debugger. GDB has tons of features, however, you only need to use a few for it to be very helpful. There is complete documentation for GDB online, or you can read the man page, and the quick reference sheet is very handy.

Also, if you did not pick one up, this reference card is useful when first learning the gdb commands. It's in postscript, so you can print it out with lpr or use ghostview.

Basic features of a debugger

When you are execute a program that does not behave as you like, you need some way to step through you logic other than just looking at your code. Some things you want to know are:

* What statement or expression did the program crash on?
* If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters?
* What are the values of program variables at a particular point during execution of the program?
* What is the result of a particular expression in a program?

Starting GDB

You need to tell the lcc compiler that you plan to debug your program. You use the -g flag to do this. The command will now look like

lcc -g trees.c

which will create the a.out executable. To run this under the control of gdb, you type

gdb a.out

When gdb starts, your program is not actually running. It won't run until you tell gdb how to run it. Whenever the prompt appears, you have all the commands on the quick reference sheet available to you.

* run command-line-arguments

Starts your program as if you had typed

a.out command-line arguments

or you can do the following

a.out < somefile

to pipe a file as standard input to your program

* break place

Creates a breakpoint; the program will halt when it gets there. The most common breakpoints are at the beginnings of functions, as in

(gdb) break Traverse

Breakpoint 2 at 0x2290: file main.c, line 20

The command break main stops at the beginning of execution. You can also set breakpoints at a particular line in a source file:

(gdb) break 20

Breakpoint 2 at 0x2290: file main.c, line 20

When you run your program and it hits a breakpoint, you'll get a message and prompt like this.

Breakpoint 1, Traverse(head=0x6110, NumNodes=4)

at main.c:16

* delete N

Removes breakpoint number N. Leave off N to remove all breakpoints. info break gives info about each breakpoint

* help command

Provides a brief description of a GDB command or topic. Plain help lists the possible topics

* step

Executes the current line of the program and stops on the next statement to be executed

* next

Like step, however, if the current line of the program contains a function call, it executes the function and stops at the next line.

* step would put you at the beginning of the function

* finish

Keeps doing nexts, without stepping, until reaching the end of the current function

* Continue

Continues regular execution of the program until a breakpoint is hit or the program stops

* file filename

Reloads the debugging info. You need to do this if you are debugging under emacs, and you recompile in a different executable. You MUST tell gdb to load the new file, or else you will keep trying to debug the old program, and this will drive you crazy

* where

Produces a backtrace - the chain of function calls that brought the program to its current place. The command backtrace is equivalent

* print E

prints the value of E in the current framein the program, where E is a C expression (usually just a variable). display is similar, except every time you execute a next or step, it will print out the expression based on the new variable values

* quit

Leave GDB. If you are running gdb under emacs,