Time in Unix

- Unix time, or POSIX time, is a system for describing points in time, defined as the number of seconds elapsed since midnight proleptic Coordinated Universal Time (UTC) of January 1, 1970, not counting leap seconds. It is widely used not only on Unix-like operating systems but also in many other computing systems and file formats.
- Example: 1264516089 (2010-01-26 14:28:09Z)
- On February 13, 2009 at exactly 23:31:30 (UTC), the decimal representation of Unix time was equal to '1234567890'. Parties and other celebrations were held around the world, among various technical subcultures, to celebrate the 1234567890 day.[1][2][3]

Useful library calls

- #include <sys/time.h>

- int gettimeofday(struct timeval *tv, struct timezone *tz)
  - struct timeval {
  - time_t tv_sec; /* seconds */
  - suseconds_t tv_usec; /* microseconds */
  - };;

  - On linux, the tz parameter is always set to NULL
  - Example
    - struct timeval curTime;
    - (void) gettimeofday(&curTime, (struct timezone *)0);

- char *ctime(const time_t *timep)
  - Example
    - (void) gettimeofday(&curTime, (struct timezone *)0);
    - sprintf(tmpLine,"%s ", (char *)ctime((const long int *)&curTime));
    - //remove the CR that ctime adds
    - tmpLine[strlen(tmpLine)-2] = 0x0;
    - tmpLine[strlen(tmpLine)-1] = 0x0;
    - ;

Computing time measurements (e.g., delays)

```
double getTime()
{
    struct timeval curTime;
    (void) gettimeofday (&curTime, (struct timezone *) NULL);
    return (((((double) curTime.tv_sec) * 1000000.0)
            + (double) curTime.tv_usec) / 1000000.0);
}
```

Example: setting a timeout on a socket call

```
alarm(2) //set the timeout for 2 seconds
rc = recv()                    //do a socket read
If (rc == -1)
  if (errno == EINTR)          //If no data is available, it unblocks with error
                               //2 seconds
        timeouts++;

alarm(0);                      //turn off alarm
```

Unix Processes

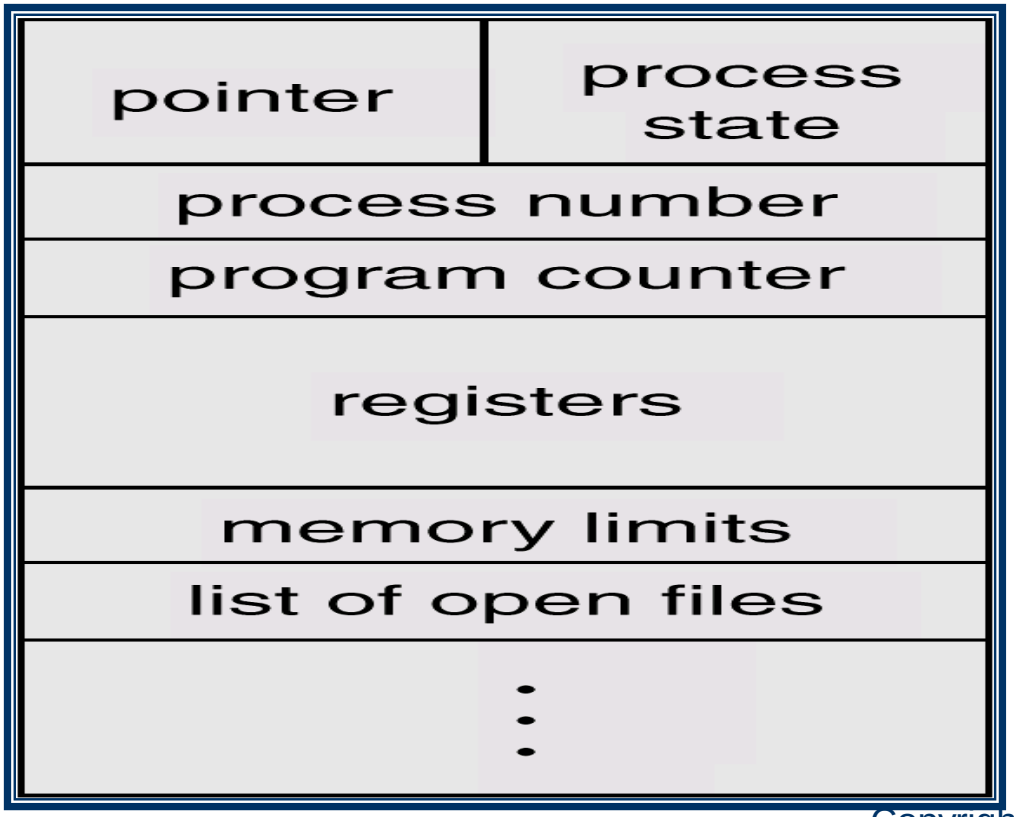
- A process is a program in execution.
  - The unit of work in a time-sharing system.
- A process has:
  - A text section: code
  - A data section: global data
  - A stack section: temp data
  - State: defined by the current activity
    - New, running, waiting, ready, terminated

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.

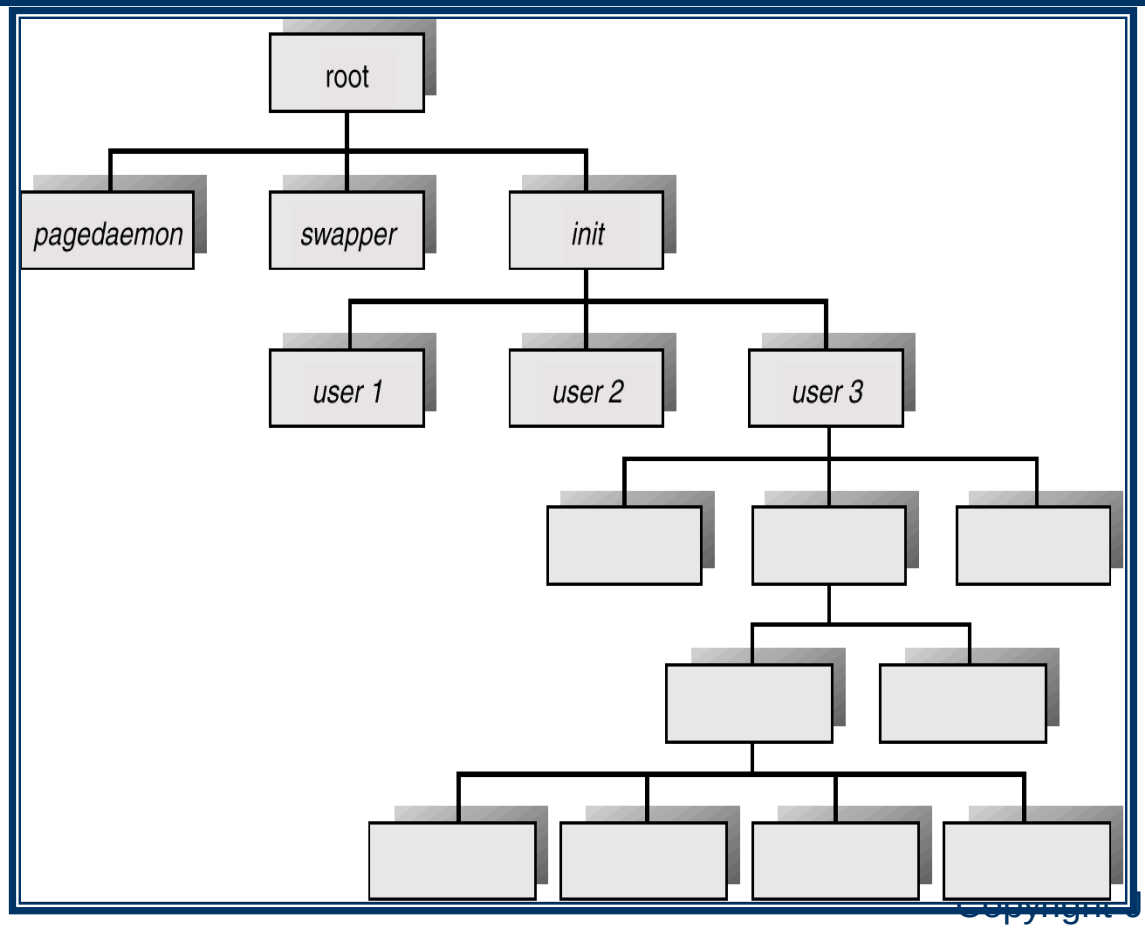
Process Creation

- 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)) {
        }
```

```
#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);
  }
}
```

malloc/free System Calls

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

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 needto 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 sheetis 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, soyou 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 justlooking 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,