Files

How to write a file copy program in standard C?

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fclose(FILE *fp);
```

We can also use character-based functions such as:

```
#include <stdio.h>
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

With either approach, we can write a C program that will work on any operating system as it is in standard C.
What is the difference between the two approaches?

```c
/* lab/stdc-mycp.c */
/* appropriate header files */
#define BUF_SIZE 65536

int main(int argc, char *argv[])
{
    FILE *src, *dst;
    size_t in, out;
    char buf[BUF_SIZE];
    int bufsize;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s <buffer size> <src> <dest>\n", argv[0]);
        exit(1);
    }
    bufsize = atoi(argv[1]);
    if (bufsize > BUF_SIZE) {
        fprintf(stderr,"Error: %s: max. buffer size is %d\n",argv[0], BUF_SIZE);
        exit(1);
    }
    src = fopen(argv[2], "r");
    if (NULL == src) exit(2);

    dst = fopen(argv[3], "w");
    if (dst < 0) exit(3);

    while (1) {
        in = fread(buf, 1, bufsize, src);
        if (0 == in) break;
        out = fwrite(buf, 1, in, dst);
        if (0 == out) break;
    }

    fclose(src);
    fclose(dst);
    exit(0);
}
```

POSIX/Unix File Interface

The system call interface for files in POSIX systems like Linux and MacOSX.
A *file* is a named, ordered stream of bytes.

- `open(..)` Open a file for reading or writing. Also allows a file to be locked providing exclusive access.
- `close(..)`
- `read(..)` The read operation is normally *blocking*.
- `write(..)`
- `lseek(..)` Seek to an arbitrary location in a file.
- `ioctl(..)` Send an arbitrary control request (specific to a device). e.g. rewinding a tape drive, resizing a window etc.

Let's rewrite the file copy program using the POSIX system call interface.

```
/* A simple file copy program  */
/* lab/files-processes/mycp.c */
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#define MODE 0666
#define BUF_SIZE 8192

void main(int argc, char *argv[]) {
    int src, dst, in, out;
    char buf[BUF_SIZE];
    if (argc != 3) exit(1);
    src = open(argv[1], O_RDONLY); if (src < 0) exit(2);
    dst = creat(argv[2], MODE); if (dst < 0) exit(3);

    while (1) {
        in = read(src, buf, BUF_SIZE);
        if (in <= 0) break;
        out = write(dst, buf, in);
        if (out <= 0) break;
    }
    close(src); close(dst); exit(0);
}
```

Effect of buffer size on I/O speed

- Observe the effect of buffer size on the speed of the copying. Experiment using the file copy program with different buffer sizes on a large file and time the copy.
- Buffering helps adjust the data rate between two entities to avoid overflow.
- Buffering can also improve performance of systems by allowing I/O to happen ahead of time or to have I/O happen in parallel with computing.
- *Buffering is a widely used concept in Computer Science.*

Effect of Buffering on File I/O (System Calls)

The following times are for the file copy program with varying buffer sizes. All times are in seconds. Total speedup due to buffering is 1455!

buffer size	elapsed	user	system
1	36.387	1.565	34.398
2	17.783	0.757	16.974
4	9.817	0.400	9.393
8	4.603	0.180	4.375
16	2.289	0.093	2.190
32	1.142	0.047	1.091
64	0.581	0.017	0.562
128	0.299	0.012	0.286
256	0.158	0.007	0.150
512	0.090	0.003	0.084
1024	0.054	0.001	0.051
2048	0.035	0.001	0.032
4096	0.025	0.000	0.024

Do we get a similar improvement with the standard C program? Why or why not?

Effect of Buffering on File I/O (Standard C)

The following times are for the file copy program with varying buffer sizes. All times are in seconds. Total speedup is around 40.

buffer size	elapsed	user	system
1	0.965	0.945	0.018
2	0.502	0.475	0.026
4	0.267	0.244	0.021
8	0.156	0.136	0.019
16	0.082	0.053	0.028
32	0.056	0.032	0.023
64	0.042	0.014	0.027
128	0.034	0.012	0.021
256	0.033	0.010	0.022
512	0.029	0.008	0.021
1024	0.029	0.006	0.022
2048	0.028	0.004	0.023
4096	0.024	0.001	0.022

Why does the standard C program behave differently?

Script for testing effects of buffering

```
#!/bin/sh
/bin/rm -f mycp.log
for i in 1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384
do
    echo -n "buffer size = " $i  >> mycp.log
    (time mycp $i test3.data junk)  &>>  mycp.log
    echo >> mycp.log
    /bin/rm -f junk
done
```

lab/files-processes/test-mycp.sh

Examples of Buffering from "Real-Life"

- Ice cube trays. You have one tray that you get ice cubes from and another full tray that is not used. When the first tray is empty, you refill that tray and let it freeze while you get ice cubes from the other tray.
- Shock absorbers in car, truck or mountain bike.
- Ski lift is a circular buffer
- Two parents buffer a child's demand for attention.
- Multiple elevators in a hotel lobby. An escalator might be considered a circular buffer.
- Traffic lights at an intersection buffer the flow of traffic through the limited resource that is an intersection. A round-about is a circular-buffer solution to the same problem.
- Formula One tire changing. Person A holds a new tire, person B sits in place with the torque wrench. There is usually a person C who collects the old tire – depends on the team and pit lane. When the car stops and is raised up, person B undoes the tire, person C removes it, person A puts the new tire in place, person B torques it and then raises his hand. The whole operation typically is done in under 3 seconds The operation can be thought of as triple buffering.

MS Windows File Interface

A *file* is a named, ordered stream of bytes.

- `OpenFile() or CreateFile(..)` Open or create a file for reading or writing. Returns a HANDLE (reference) toa file structure used to identify the file for other system calls.
- `CloseHandle(..)`
- `ReadFile(..)` The read operation is normally *blocking*.
- `WriteFile(..)`
- `SetFilePointer(..)` Seek to an arbitrary location in a file.

Processes

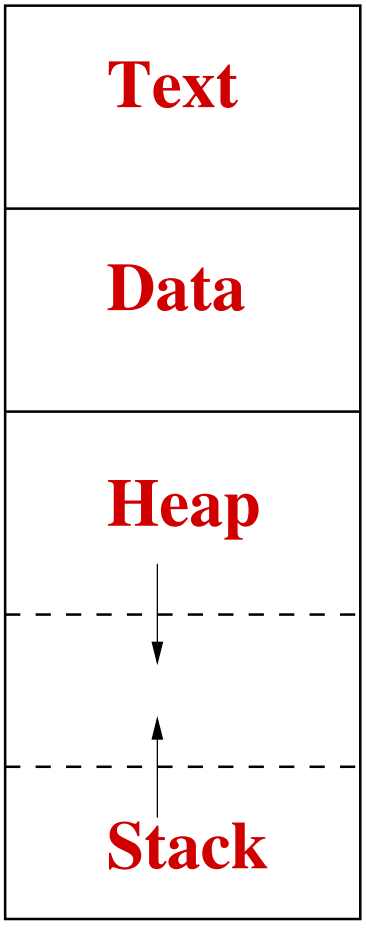
Components of a process:

- the executable code (also known as program text or text segment)
- the data on which the program will execute
- status of the process
- resources required by the process: e.g. files, shared libraries etc.

The data associated with a process is divided into several segments:

- Global and static variables: data segment
- Dynamically allocated variables: heap segment
- Local variables, function/method arguments and return values: stack segment

The Linux/UNIX Process model



0x00000000

Program Binary

Global/Static variables

Dynamically allocated variables

Local variables, function/method arguments

0xFFFFFFFF

Creation of Processes

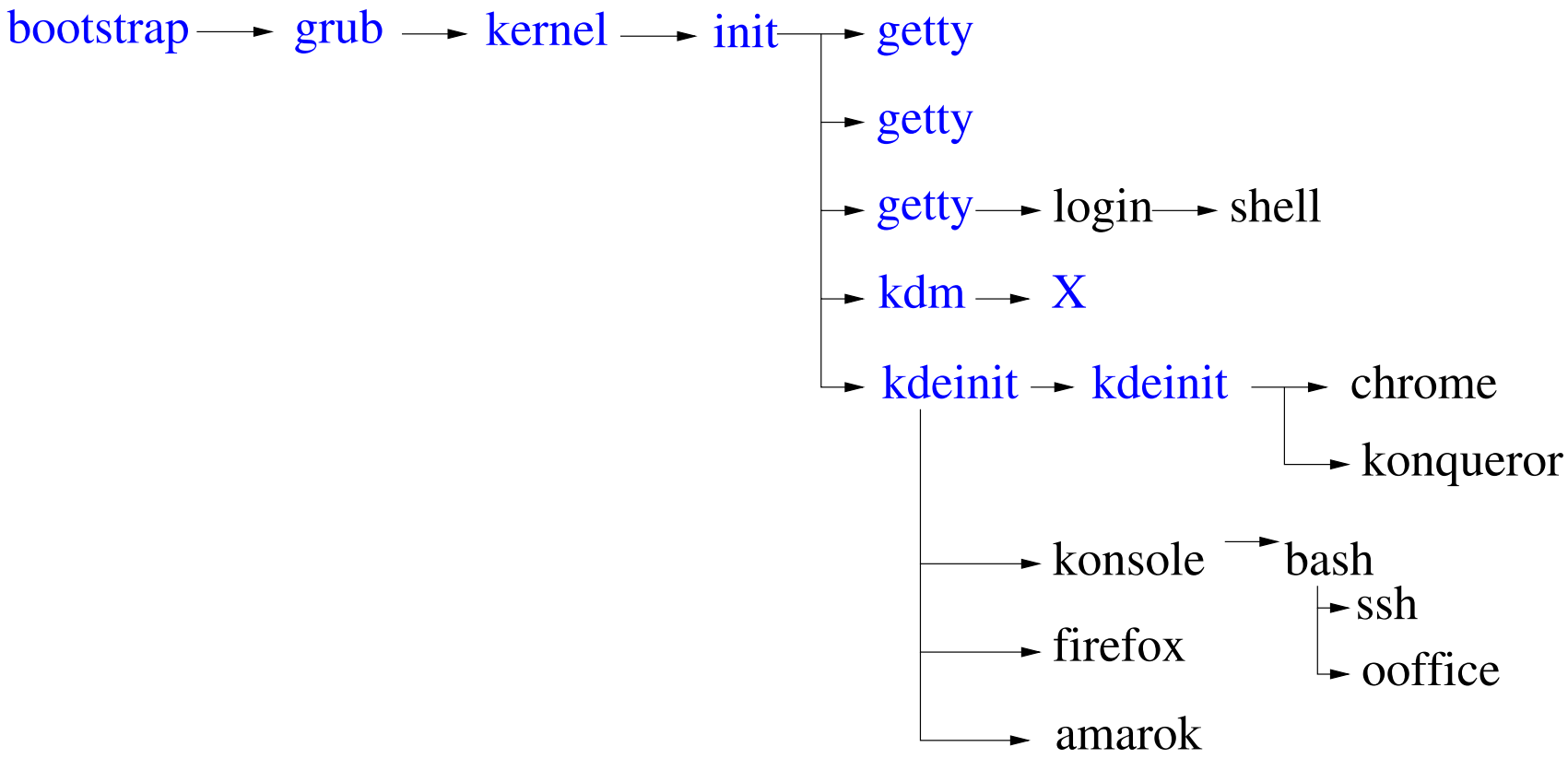
- To the user, the system is a collection of processes. Some of them are part of the operating system, some perform other supporting services and some are application processes.
- Why not just have a single program that does everything?
- Multiplicity example.
- How is a process created?
- How is the operating system created?
  Let's examine the bootup of Linux and Microsoft Windows.

Initializing the Operating System

"Booting" the computer.

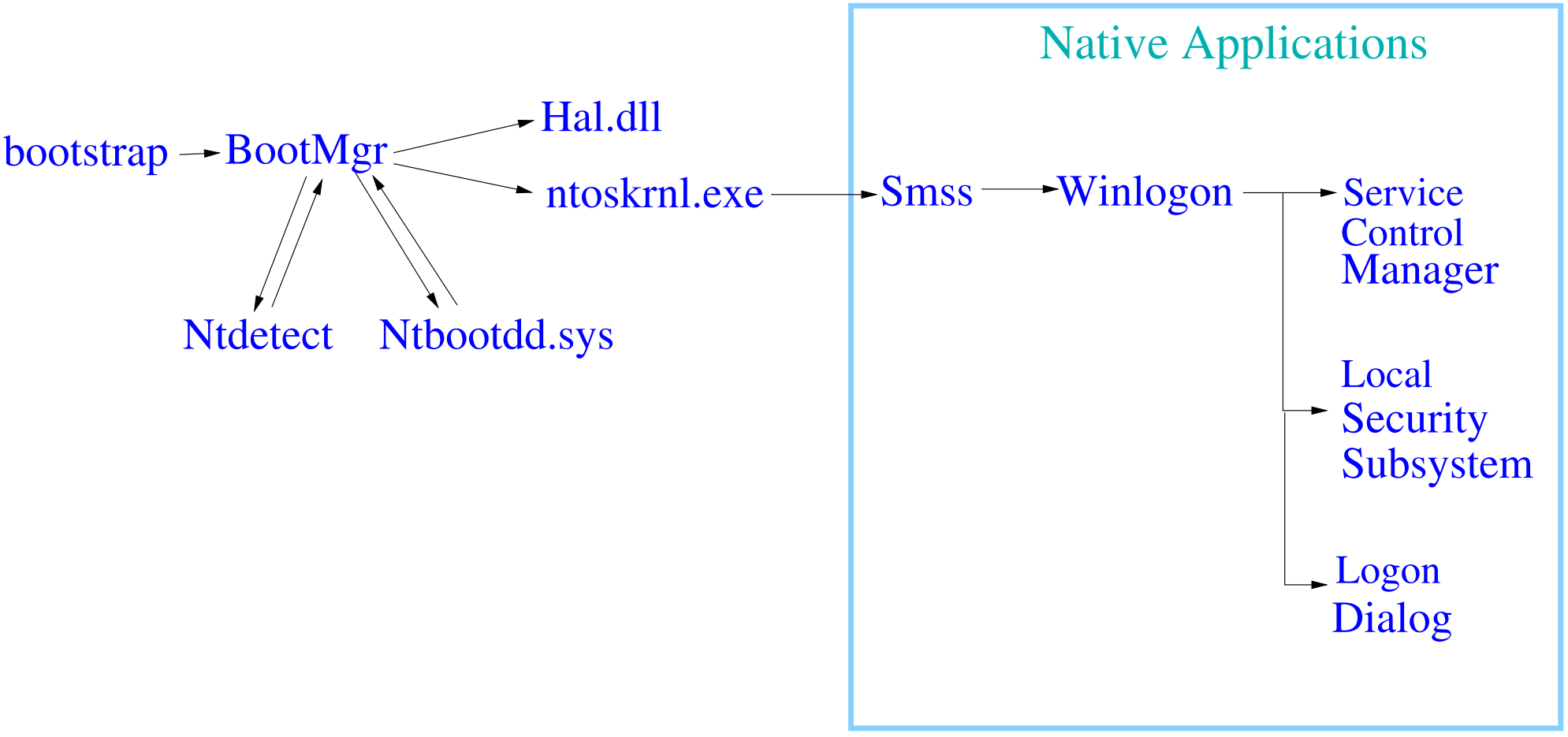
```
Main Entry: bootï£¡strap
Function: noun
Date: 1913
1 plural: unaided efforts -- often used in the phrase by
          one's own bootstraps
2 : a looped strap sewed at the side or the rear top of a boot
    to help in pulling it on.
```

The Linux Bootup Process



init has been replaced by systemd in some newer distributions

Microsoft Windows Bootup Process



Smss: Session Manager SubSystem

Executing Computations

- The Unix model (followed by most operating systems) is to create a new process every time to execute a new computation. The system at any time looks like a tree of processes, with one process being the ancestor of all other processes.
- *What's the advantage of creating a process each time we start a new computation?*

The fork() system call

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

pid_t fork(void);
```

- The fork() system call creates a child process that is a clone of the parent. The stack, data and heap segments are the same at the moment of creation. The program text is also logically copied (but may be physically shared).
- The child process differs from the parent process only in its process id and its parent process id and in the fact that resource utilization is set to zero.
- One easy way to communicate between a parent and a child is for the parent to initialize variables and data structures before calling `fork()` and the child process will inherit the values.
- The `fork()` is called once but it returns twice! (one in the parent process and once in the child process)

Two roads diverged in a wood, and I–
I took the one less traveled by,
and I got lost!

wait and waitpid system calls

- Used to wait for a state change in a child process.

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- The `waitpid()` system call suspends the calling process until one of its child terminates or changes states. Using -1 for `pid` in `waitpid()` means to wait for any of the child processes. Otherwise, `pid > 0` provides the specific process id to wait for.
- It is possible to do a non-blocking wait using the `WNOHANG` option, as shown below where the call will return immediately if no child is done or changed state:

```
waitpid(-1, &status, WNOHANG);
```

- The `wait()` system call suspends execution of the calling process until one of its children terminates. The call `wait(&status)` is equivalent to:

```
waitpid(-1, &status, 0);
```

- Check the man page on how to check the `status` variable to get more information about the child process whose pid was returned by `waitpid()` or `wait()` call.

```c
/* lab/files-processes/fork-and-wait.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
void childs_play(), err_sys(char *msg);
int main(void) {
    pid_t   pid;
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {        /* child */
                childs_play(); exit(0);
         }
    /* parent continues concurrently with child */
    printf("Created child with pid %d\n",pid);
    sleep(2);
    printf("Shoo away!\n");
    /* wait for normal termination of child process */
    if (waitpid(pid, NULL, 0) != pid)
            err_sys("waitpid error");
    exit(0);
}
void childs_play() {printf("Hey, I need some money! \n");}

void err_sys(char *msg) {
    fprintf(stderr, msg);
    fflush(NULL); /* flush all output streams */
    exit(1); /* exit abnormally */
}
```

```c
/*  lab/files-processes/fork-hello-world.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void print_message_function( void *ptr ), err_sys(char *msg);
int main(void)
{
    pid_t   pid;
    char *message1 = "Goodbye";
    char *message2 = "World";

    printf("before fork\n");
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {        /* first child */
                print_message_function(message1);
                sleep(2); exit(0);
         }
    printf("Created child: pid=%d\n",pid);
    /* parent continues and creates another child */
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {        /* second child */
                print_message_function(message2);
                sleep(2); exit(0);
         }
    printf("Created child: pid=%d\n",pid); /* parent */
    sleep(2);
    exit(0);
}
void print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s ", message);
}
void err_sys(char *msg)
{
    fprintf(stderr, msg);
    fflush(NULL); /* flush all output streams */
    exit(1); /* exit abnormally */
}
```

```
/* lab/files-processes/fork-child-grandchild.c */
/* include statements, prototypes, blah blah */
int main(void)
{
  pid_t   pid;

  printf("original process, pid = %d\n", getpid());
  if ((pid = fork()) < 0)
      err_sys("fork error");
  else if (pid == 0) {          /* child */
      printf("child = %d, parent = %d\n",
                                getpid(), getppid());
      if ( (pid = fork()) < 0)
          err_sys("fork error");
      else if (pid == 0) {    /* grandchild */
          printf("grandchild = %d, parent = %d\n",
                                getpid(), getppid());
          exit(0);
          }
      /* child waits for the grandchild */
      if (waitpid(pid, NULL, 0) != pid)
          err_sys("waitpid error");
      exit(0); /* the child can now exit */
      }
  /* original process waits for its child to finish */
  if (waitpid(pid, NULL, 0) != pid)
      err_sys("waitpid error");
  exit(0);
}
```

The exec() system call

```
#include <unistd.h>
int  execve(const char *filename, char *const argv [], char *const envp[]);
```

- The `execve()` executes the program pointed to by the `filename` parameter. It does not return on success, and the text, data and stack segments of the calling process are overwritten by that of the program loaded. The program invoked inherits the calling process's process id. See the man page for more details.
- The `execve()` function is called once but it never returns on success! The only reason to return is that it failed to execute the new program.
- The following variations are front-ends in the C library. In the first two variations, we only have to specify the name of the executable (without any '/') and the function searches for its location in the same way as the shell using the `PATH` environment variable. In the last three variations we must specify the full path to the executable.

```
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
int execl(const char *path, const char *arg, ...);
int execle(const char *path, const char  *arg, ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
```

How does the shell find an executable?

- When we type the name of a program and hit enter in the shell, it searches for that executable in a list of directories specified usually by the `PATH` environment variable
- We can check the value of the PATH variable with the `echo` command:

```
[amit@onyx ~]$ echo $PATH
/bin:/usr/lib64/ccache:/usr/local/bin:/usr/bin:/home/faculty/amit/bin:.:
```

  We get a colon separated list of directories. The search is done in order from the first to the last directory in the list and chooses the first instance of the executable it finds
- We can ask the shell which executable it will use with the `which` command. For example:

```
[amit@onyx ~]$ which gcc
/bin/gcc
```

- A program can find the value of the `PATH` variable with the system call `getenv("PATH")`

Example of exec'ing a process

```
/* lab/files-processes/fork-and-exec.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
void err_sys(char *msg);
int main(void)
{
    pid_t   pid;
    if ( (pid = fork()) < 0)
            err_sys("fork error");
    else if (pid == 0) {            /* child */
                    execlp("./print-pid","print-pid",0);
                    err_sys("exec failed");
                    exit(1);
         }
    printf("Created child with pid %d\n",pid);
    /* parent continues concurrently with child */

    /* wait for normal termination of child process */
    if (waitpid(pid, NULL, 0) != pid)
                err_sys("waitpid error");
    exit(0);
}

void err_sys(char *msg) {
        fprintf(stderr, msg);
        fflush(NULL); /* flush all output streams */
        exit(1); /* exit abnormally */
}
```

```
/* The exec'ed program */
#include <unistd.h>
#include <stdio.h>
int main() {
        printf("after exec pid=%d\n",getpid());
        exit(0);
}
```

A simple shell

```
/* lab/files-processes/simple-shell.c, need error.c and ourhdr.h to compile */
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"
int main(void) {
    char    buf[MAXLINE];
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        buf[strlen(buf) - 1] = 0;   /* replace newline with null */
        if ( (pid = fork()) < 0)
            err_sys("fork error");
        else if (pid == 0) {        /* child */
            execlp(buf, buf, (char *) 0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }
        /* parent */
        if ( (pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

Signals: asynchronous events

Linux/Unix signals are a type of event. Signals are asynchronous in nature and are used to inform processes of certain events happening.
Examples:

- User pressing the interrupt key (usually `Ctl-c` or `Delete` key). Generates the `SIGINT` signal.
- User pressing the stop key (usually `Ctl-z`). Generates the `SIGTSTP` signal, which stops (suspends) the process.
- The signal `SIGCONT` can restart a process if it is stopped.
- Signals are available for alarm (`SIGALRM`), for hardware exceptions, for when child processes terminate or stop and many other events.
- Special signals for killing (`SIGKILL`) or stopping (`SIGSTOP`) a process. These cannot be ignored by a process.

POSIX signals list

Read `man signal` and `man 7 signal` for more information.

```
SIGHUP   Hangup detected on controlling terminal
         or death of controlling process
SIGINT   Interrupt from keyboard
SIGQUIT  Quit from keyboard
SIGILL   Illegal Instruction
SIGABRT  Abort signal from abort
SIGFPE   Floating point exception
SIGKILL  Kill signal
SIGSEGV  Invalid memory reference
SIGPIPE  Broken pipe: write to pipe with no readers
SIGALRM  Timer signal from alarm
SIGTERM  Termination signal
SIGUSR1  User-defined signal 1
SIGUSR2  User-defined signal 2
SIGCHLD  Child stopped or terminated
SIGCONT  Continue if stopped
SIGSTOP  Stop process
SIGTSTP  Stop signal from keyboard
SIGTTIN  tty input for background process
SIGTTOU  tty output for background process
```

Other Unix signals (not part of POSIX standard)

SIGTRAP	Trace/breakpoint trap
SIGIOT	IOT trap. A synonym for SIGABRT
SIGBUS	Bus error
SIGSYS	Bad argument to routine (SVID)
SIGSTKFLT	Stack fault on coprocessor
SIGURG	Urgent condition on socket (4.2 BSD)
SIGIO	I/O now possible (4.2 BSD)
SIGPOLL	A synonym for SIGIO (System V)
SIGCLD	A synonym for SIGCHLD
SIGXCPU	CPU time limit exceeded (4.2 BSD)
SIGXFSZ	File size limit exceeded (4.2 BSD)
SIGVTALRM	Virtual alarm clock (4.2 BSD)
SIGPROF	Profile alarm clock
SIGPWR	Power failure (System V)
SIGINFO	A synonym for SIGPWR
SIGLOST	File lock lost
SIGWINCH	Window resize signal (4.3 BSD, Sun)
SIGUNUSED	Unused signal

Signals (contd.)

- For each signal there are three possible actions: default, ignore, or catch. The system call `signal()` attempts to set what happens when a signal is received. The prototype for the system call is:

```
void (*signal(int signum, void (*handler)(int)))(int);
```

- The above prototype can be made easier to read with a typedef as shown below.

```
typedef void sighandler_t(int);
sighandler_t *signal(int, sighandler_t *);
```

- The header file <signal.h> defines two special dummy functions `SIG_DFL` and `SIG_IGN` for use as signal catching actions. For example:

```
signal(SIGALRM, SIG_IGN);
```

To kill or to really kill?

- The system call `kill()` is used to send a specified signal to a specified process. For example:

```
kill(getpid(), SIGSTOP);
kill(getpid(), SIGKILL);
kill(pid, SIGCONT);
```

- Special signals for killing (`SIGKILL`) or stopping (`SIGSTOP`) a process. These cannot be ignored by a process.
- Linux has a command named `kill` that invokes the `kill()` system call.

```
kill -s signal pid
kill -l  --> list all signals
kill -9  --> send SIGKILL
```

- To kill a process use `kill -1` (SIGHUP) or `kill -15` (SIGTERM) first to give the process a chance to clean up before being killed (as those signals can be caught). If that doesn't work, then use `kill -9` to send SIGKILL signal that cannot be caught or ignored. In some circumstances, however, even SIGKILL doesn't work....

kill -9

Because I could not stop for Death,
He kindly stopped for me;
The carriage held but just ourselves
And Immortality.
...

Emily Dickinson

A bigger example of systems programming

```
/* lab/files-processes/timeout.c: set time limit on a process.
   Usage: timeout [-10] command
*/
#include <stdio.h>
#include <signal.h>
#include <sys/wait.h>
int pid; /* child process id */
char *progname;
static void onalarm(int signo);

int main(int argc, char *argv[])
{
    void error(char *msg, char *arg);
    int sec=10, status;

    progname = argv[0];
    if (argc > 1 && argv[1][0] == '-') {
        sec = atoi(&argv[1][1]); argc--; argv++;
    }

    if (argc < 2) {
        error("Usage: %s [-10] command", progname);
    }
```

continued on next slide...

```
  /* main function continued */
  if ((pid=fork()) == 0) {
      execvp(argv[1], &argv[1]);
      error("couldn't start %s", argv[1]);
  }
  signal(SIGALRM, onalarm);
  alarm(sec);
  if ((wait(&status) == -1) || WIFSIGNALED(status))
      error("%s killed", argv[1]);
  exit(WEXITSTATUS(status));
}

/* kill child process when alarm arrives */
void onalarm(int signo)
{
    kill(pid, SIGKILL);
}

void error(char *msg, char *arg)
{
    fprintf(stderr, msg, arg);
    fprintf(stderr,"\n");
    exit(1);
}
```

Pedal to the metal: fork test

```
/* lab/files-processes/fork-test.c: Try to generate a lot of processes */
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
void err_sys(char *msg);
#define MAXNUM 258
int main(void) {
 pid_t   pid;
 int i;
 for (i=0; i<MAXNUM; i++) {
   if ( (pid = fork()) < 0)
     err_sys("fork error");
   else if (pid == 0) {        /* ith child */
     sleep(20); exit(0);
   }
   printf("Created child %d with pid %d\n",i,pid);
 }
 exit(0);
}
void err_sys(char *msg) {
 fprintf(stderr, msg);
 if (errno == EAGAIN)
   fprintf(stderr, "\n Cannot create a task structure\n");
   if (errno == ENOMEM)
     fprintf(stderr, "\n Not enough memory\n");
   fflush(NULL); /* flush all output streams */
   exit(1); /* exit abnormally */
}
```

Pipes

- A pipe allows communication between two processes that have a common ancestor.
- A pipe is a half-duplex (data flows in only one direction) FIFO buffer with an API similar to file I/O.

```
#include <unistd.h>

int pipe(int filedes[2]);
// returns filedes[0] for reading, filedes[1] for writing.
```

- Reading from a pipe whose write end has been closed causes an End Of File to be returned. Writing to a pipe whose read end has been closed causes the signal `SIGPIPE` to be generated. The write returns with `errno` set to `EPIPE`.
- The size of pipe is limited to `PIPE_BUF`. A write of `PIPE_BUF` or less will not interleave with the writes from other processes. The constant `PIPE_BUF` is defined in the file `/usr/include/linux/limits.h`

The Power Of Pipelines

Find the 10 most frequent words in a given text file (and their respective counts).

```
cat Shakespeare.txt | tr -cs "[A-Z][a-z]['’]" "[\012*]" | tr A-Z a-z |
sort | uniq -c | sort -rn | sed 10q
```

Hello World with a Pipe

```
/* lab/files-and-processes/hello-pipe.c */
/* appropriate header files */

int main(void)
{
    int     n, fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0)
        err_sys("fork error");

    else if (pid > 0) {     /* parent */
        close(fd[0]); /* close read end of pipe */
        write(fd[1], "hello world\n", 12);

    } else {                /* child */
        close(fd[1]); /* close write end of pipe */
        n = read(fd[0], line, MAXLINE);
        printf("child read %d characters from the parent in the pipe: %s", n, line);
    }
    exit(EXIT_SUCCESS);
}
```

Named Pipes (FIFOs)

- Named Pipes (FIFOs) allow arbitrary processes to communicate.

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

int mkfifo ( const char *pathname, mode_t mode );

mkfifo pathname
```

- If we write to a FIFO that no process has open for reading, the signal `SIGPIPE` is generated. When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.
- The reads/writes can be made blocking or non-blocking.
- If we have multiple writers for a FIFO, atomicity is guaranteed only for writes of size no more than `PIPE_BUF`.

Uses of FIFOs

- Can be used by shell commands to pass data from one shell pipeline to another, without creating intermediate temporary files.

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

  Another example of a nonlinear pipeline:

```
wc < fifo1 &
cat /usr/share/dict/words | tee fifo1 | wc -l
```

- Look at examples:
  `lab/files-and-processes/hello-fifo.c`
  `lab/files-and-processes/fifo-talk.c`

Client Server Communication Using FIFOs

- The server creates a FIFO using a pathname known to the clients. Clients write requests into this FIFO.
- The requests must be atomic and of size less than `PIPE_BUF`, which is defined in `limits.h` standard header file.
- The server replies by writing to a client-specific FIFO. For example, the client specific FIFO could be `/tmp/serv1.xxxxx` where `xxxxx` is the process id of the client.

System Calls Introduced

- exit()
- `open()`, `creat()`, `close()`, `read()`, `write()`
- `fork()`
- `wait()`, `waitpid()`
- `execvp()`, `execlp()`
- `alarm()`
- `signal()`
- `getpid()`, `getppid()`
- `sleep()`, `kill()`
- `pipe()`, `mkfifo()`

Exercises

1. **Passing data to your progeny?** Write a program that creates and fills some data structure (like an array). Then it forks a child process. Check if the child process inherits the initialized data structure. What about opening a file in the child process that was opened in the parent process before the fork?
2. **Waiting for Godot!** Write a program that creates as many processes as the number of CPUs on your system (using the fork system call). Each created process generates one billion random integers using the `random()` function. Set the seed on each process using `srandom()` with the process id (obtained using `getpid()`). Use the random numbers for something: for example, count how many numbers were within the range 90..110 or some other inane property. After generating one billion random numbers, each process sleeps for 10 seconds. This behavior is repeated in an infinite loop. Watch the load using the system monitor. The number of CPUs can be determined via the following system call:
   `sysconf(_SC_NPROCESSORS_CONF)`
3. **Struck by lightning while waiting for Godot!**. Take the same program and add an alarm interval as a command line argument, where `godot` is the name of the program executable:
   `godot <alarm interval>`
   The program now sets up a signal handler and an alarm. When alarm arrives, it kills all the running child processes using the `kill()` system call. Also add a loop at the end of the main program that uses the `waitpid()` system call to determine what signal caused the child process to terminate. Note that the alarm timer isn't inherited by the child processes.

Exercises (contd.)

4. **Walkie-Talkie**. Write a program that uses two pipes to allow two-way communication between a parent and child process. May be they can finally have a real conversation....
5. **Chit Chat**. Write a server program that creates two named pipes (FIFOs) and then waits for a client program to write a request to one of the named pipes. Then it uses the other named pipe to reply to the client. Since we are using named pipes, you will be able to run these two programs in two separate windows and watch them communicate!
6. **Multi-process Chat Server**. This generalizes the solution from the last problem. Write a multi-process server that can chat with multiple clients simultaneously by forking multiple copies of itself. Also develop a client program to test the server. This will use named pipes for the communication.

MS Windows API for Processes

In MS Windows, the system call interface is not documented. Instead the MS Windows API is documented, which helps with being able to run programs portably across multiple versions of the MS Windows operating systems.
Creating a process gives a *handle* that is used to refer to the actual object that represents a process/thread.

- CreateProcess(...). Fork-and-exec a new process.
- CloseHandle(...).
- ExitProcess(...), TerminateProcess(...), GetExitCodeProcess(...), GetCurrentProcessId(), GetCurrentProcess().
- WaitForSingleObject(...), WaitForMultipleObjects(...). These can be used to wait for either a process or a thread.

Get detailed information from `http://msdn.microsoft.com/library/`

CreateProcess Call in MS Windows API

```
BOOL WINAPI CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Processes Related Calls in MS Windows API

```
WaitForSingleObject(hProcess, INFINITE);

CloseHandle(pi.hProcess);

DWORD WINAPI GetCurrentProcessId(void);
HANDLE WINAPI GetCurrentProcess(void);

VOID WINAPI ExitProcess(
  UINT uExitCode
);
BOOL WINAPI TerminateProcess(
  HANDLE hProcess,
  UINT uExitCode
);
BOOL WINAPI GetExitCodeProcess(
  HANDLE hProcess,
  LPDWORD lpExitCode
);
```

MS Windows API for Processes

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD dwProcessId;
  DWORD dwThreadId;
} PROCESS_INFORMATION,
 *LPPROCESS_INFORMATION;

typedef struct _SECURITY_ATTRIBUTES {
  DWORD nLength;
  LPVOID lpSecurityDescriptor;
  BOOL bInheritHandle;
} SECURITY_ATTRIBUTES,
 *LPSECURITY_ATTRIBUTES;
```

Checking Errors in System Calls

- `DWORD GetLastErrorCode(void)`. Retrieves the calling thread's last-error code value. The last-error code is maintained on a per-thread basis. Multiple threads do not overwrite each other's last-error code. This function should be called right after a system call returns an error (usually we know that from a negative return value from the system call).
- To obtain an error string for system error codes, use the `FormatMessage` function.

```
DWORD FormatMessage(
  DWORD dwFlags,
  LPCVOID lpSource,
  DWORD dwMessageId,
  DWORD dwLanguageId,
  LPTSTR lpBuffer,
  DWORD nSize,
  va_list* Arguments
);
```

Sample Error Code

```
void ErrSys(char *szMsg)
{
    LPVOID lpMsgBuf;

    // Try to format the error message from the last failed call
    //  (returns # of TCHARS in message -- 0 if failed)
    if (FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | // source and processing options
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,                                // message source
        GetLastError(),                      // message identifier
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),  // language (Default)
        (LPTSTR) &lpMsgBuf,                          // message buffer
        0,    // maximum size of message buffer
              // (ignored with FORMAT_MESSAGE_ALLOCATE_BUFFER set)
        NULL  // array of message inserts
    ))
    {
      // Display the formatted string with the user supplied string at front.
      fprintf(stderr, "%s: %s\n", szMsg, (LPSTR)lpMsgBuf);
      LocalFree(lpMsgBuf); // Free the buffer.
    } else {
        fprintf(stderr, "%s: Could not get the error message!\n",  szMsg);
    }
    fflush(NULL); /* flush all output streams */
    ExitProcess(1); /* exit abnormally */
}
```

Using MS Visual Studio

- Visual Studio is available via the Dream Spark program from the college.
- Start up Visual Studio. Choose *New Project* → *Visual C++* → *Win32* → *Win32 Console Project*.
- In the Wizard window, choose *Application Settings* → *Empty Project* → *Finish*.
- Right click on the project in the right pane (*Solution Explorer*) and then choose *Add* → *Add Existing Item...*. Note that this doesn't copy the file into the Visual Studio project folder.
- Also note that, Visual Studio uses Unicode by default. For now, we will simply turn this off. Press *ALT+F7* to open the project properties, and navigate to *Configuration Properties* → *General*. Switch *Character Set* to *Multi-Byte Character Setting* from the drop-down menu.
- **Tip**. If you want to know definition of MS Windows API typedefs, right-click on the type (e.g. LPVOID) and select "go to definition" from the drop down menu.

MS Windows API Examples

- `lab/ms-windows/files-processes/fork-and-exec.c`
- `lab/ms-windows/files-processes/fork-and-wait.c`
- `lab/ms-windows/files-processes/fork-hello-world.c`
- `lab/ms-windows/files-processes/fork-test.c`
- `lab/ms-windows/files-processes/file-copy.c`
- and others in the ms-windows/files-processes examples folder....

Microsoft PowerShell

Powershell is a shell with a command-line and scripting language available on Microsoft platforms.

- Aliases are built-in for common commands used in bash with Unix/Linux/Mac OSX systems. For example, TAB is used for command completion and aliases exist for `ls`, `cp`, `man`, `date` etc.
- Pipes are also supported but they pass objects instead of unstructured text streams.
- Includes a dynamically typed scripting language with .NET integration. Here is a simple example of a loop:

  ```
  while ($true) {.\fork-hello-world; echo ""}
  ```

- The following shows how to time a command or script in powershell:

  ```
  Measure-Command {sleep 2}
  ```

- Powershell script files are text files with a `.ps1` extension. By default, you cannot run scripts unless they are signed. To enable it, use the following command:

  ```
  Set-ExecutionPolicy RemoteSigned
  ```

  Use the command Get-Help About_Signing to learn more about signing.

Exercises

1. Setup Visual Studio and play with the examples from the class repository to familiarize yourself with the environment.
2. **Waiting for Godot!** Write a program that creates as many processes as the number of CPUs on your system (using fork system call). Each created process generates one billion random numbers and then sleeps for 5 seconds. This behavior is repeated in an infinite loop. Watch the load using the task manager. The number of CPUs can be determined via the following MS Windows API call: `GetSystemInfo(...)`