Parameter Passing in C

- In C, function parameters are passed by value
  - Each parameter is copied
  - The function can access the copy, not the original value

```c
#include <stdio.h>

void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int x = 9;
    int y = 5;
    swap(x, y);
    printf("x=%d y=%d\n", x, y);
    return 0;
}
```
Parameter Passing in C

- To pass parameters by reference, use pointers
  - The pointer is copied
  - But the copy still points to the same memory address

```c
#include <stdio.h>

void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int x = 9;
    int y = 5;
    swap(&x, &y);
    printf("x=%d y=%d\n", x, y); /* This will print: x=5 y=9 */
    return 0;
}
```

Pointer Arithmetic

- Pointers are just a special kind of variable
- You can do **calculations** on pointers
  - You can use +, -, ++, -- on pointers
  - This has no equivalent in Java
- Be careful, operators work with the size of variable types!

```c
int i = 8;
int *p = &i;
p++; /* increases p with sizeof(int) */

char *c;
c++; /* increases c with sizeof(char) */
```
Pointer Arithmetic

- This is obvious when using pointers as arrays:

```c
int i;
int array[5];
int *p = array;

for (i=0;i<5;i++) {
    *p = 0;
    p++;
}
```

- int array[5];
- int *p = array;

---

Pointer Arithmetic

- int array[5];
- int *p = array;

- int array[5];
- int *p = array;

- char array[5];
- char *p = array;

- char array[5];
- char *p = array;
Exercise

- int main ()
  {
    int i, r[6] = {1, 1, 1, 0, 0, 0};
    int *ptr;
    ptr = r;
    *ptr = 10;
    *(ptr + 1) = 5;
    r[2] = *ptr;
    *(ptr++) = 20;
    ptr += 2;
    *(++ptr) = 20;
    for (i = 0; i < 6; i++)
      printf(" r[%d] = %d\n", i, r[i]);
  }

Function Pointers

- Functions are not variables but we can define pointers to functions which will allow us to manipulate functions like variables..

- int f() : a function which returns an integer
- int* f() : a function which returns a pointer to integer
- int (*f)() : a pointer to a function which returns integer
- int (*f[])() : an array of pointer to a function which returns integer
Example

```c
void sum(int a, int b) {printf("sum: %d\n", a+b);}
void dif(int a, int b) {printf("dif: %d\n", a-b);}
void mul(int a, int b) {printf("mul: %d\n", a*b);}
void div(int a, int b) {printf("div: %f\n", a/b);}

void (*p[4])(int x, int y);

int main(void)
{
    int result;
    int i=10, j=5, op;

    p[0]=&sum; /* address of sum() */
    p[1]=&dif; /* address of dif() */
    p[2]=&mul; /* address of mul() */
    p[3]=&div; /* address of div() */

    for (op=0; op<4; op++) (*p[op])(i, j);
}
```

Operator Precedence

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>() [] -&gt; .</td>
<td>left to right</td>
<td>primary expr.</td>
</tr>
<tr>
<td>++ (postfix) -- (postfix)</td>
<td>right to left</td>
<td>postfix</td>
</tr>
<tr>
<td>+ - ! ++ (prefix) -- (prefix) (type)</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+ - &lt;= &gt; &gt;=</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td>left to right</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
<td>conditional</td>
</tr>
<tr>
<td>= += -= *= /= %=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
<td>comma</td>
</tr>
</tbody>
</table>
Exercise

1. int *a[] : 

2. int (*a[]) : 

3. int* (*a)() : 

4. int* ((a())[])() : 

5. int (*(*a)[])() : 

6. int* (*(*a[])())[] : 

Solutions

int *a[] : array[] of pointer to int
int (*a[]) : pointer to array[] of int
int* (*a)() : pointer to function which returns pointer to int
int* ((a())[])() : function which returns array[] of functions that return pointer to int
int (*(*a)[])() : function which returns pointer to array of pointers to functions which return pointer to int
int* (*(*a[])())[] : array of pointer to function which returns pointer to array of pointer to int
Static Local Variables

- Declaring a static variable means it will persist across multiple calls to the function.

```c
void foo() {
    static int i=0;
    i++;
    printf("i=%d\n", i);  /* This prints the value of i on the screen */
}

int main() {
    int i;
    for (i=0; i<3; i++) foo();
}
```

This program will output this:

```
i=1
i=2
i=3
```

Dynamic Memory Management

- `malloc()` will allocate any amount of memory you want.

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

- `malloc` takes a size (in bytes) as a parameter.
  - If you want to store 3 integers there, then you must reserve `3*sizeof(int)` bytes.
- It returns a pointer to the newly allocated piece of memory.
  - It is of type `void *`, which means "pointer to anything.
  - Do not store it as a `void *`! You should "cast" it into a usable pointer:

```c
#include <stdlib.h>
int *i = (int *) malloc(3*sizeof(int));
i[0] = 12;
i[1] = 27;
i[2] = 42;
```
Exercise

```c
int main ()
{
    int x = 10;
    int *p, *q;
    q = (int *) malloc(sizeof (int));
    *q = 60;
    p = (int *) malloc(sizeof (int));
    p = q;
    free(p);
    printf ("%d %d %d\n", x, *p, *q);
    q = &x;
    x = 70;
    p = q;
    (*p)++;
    q = x + 11;
    printf ("%d %d %d\n", x, *p, *q);
}
```

Buffered I/O

- Unbuffered I/O: each read write invokes a system call in the kernel.
  - read, write, open, close, lseek

- Buffered I/O: data is read/written in optimal-sized chunks from/to disk --> streams
  - standard I/O library written by Dennis Ritchie
Standard I/O Library

- Difference from File I/O
  - File Pointers vs File Descriptors
  - fopen vs open
    - When a file is opened/created, a stream is associated with the file.
    - FILE object
      - File descriptor, buffer size, # of remaining chars, an error flag, and the like.
  - stdin, stdout, stderr defined in <stdio.h>
    - STDIO_FILENO, STDOUT_FILENO,...

Standard I/O Efficiency

- Copy stdin to stdout using:

<table>
<thead>
<tr>
<th>total time</th>
<th>kernel time</th>
</tr>
</thead>
<tbody>
<tr>
<td>fgets, fputs : 2.6 sec</td>
<td>0.3 sec</td>
</tr>
<tr>
<td>fgetc, fputc : 5 sec</td>
<td>0.3 sec</td>
</tr>
<tr>
<td>read, write : 423 sec</td>
<td>397 sec (1 char at a time)</td>
</tr>
</tbody>
</table>
Effect of Buffer Size

- `cp file1 to file2 using read/write with buffersize:
  (5 MB file)`

<table>
<thead>
<tr>
<th>buffersize</th>
<th>exec time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50.29</td>
</tr>
<tr>
<td>4</td>
<td>12.81</td>
</tr>
<tr>
<td>16</td>
<td>3.28</td>
</tr>
<tr>
<td>64</td>
<td>0.96</td>
</tr>
<tr>
<td>256</td>
<td>0.37</td>
</tr>
<tr>
<td>1024</td>
<td>0.22</td>
</tr>
<tr>
<td>4096</td>
<td>0.18</td>
</tr>
<tr>
<td>16384</td>
<td>0.18</td>
</tr>
</tbody>
</table>

Restrictions

* When a file is opened for reading and writing:
  - Output cannot be directly followed by input without an intervening `fseek`, `fsetpos`, or `rewind`
  - Input cannot be directly followed by output without an intervening `fseek`, `fsetpos`, or `rewind`
Files and Directories

Objectives

- Additional Features of the File System
- Properties of a File.

```c
struct dirent{
    ino_t    d_ino; /* i-node number */
    char d_name[];
    ....
};
```

Directories

- dirent : file system independent directory entry

```c
struct dirent{
    ino_t d_ino;
    char d_name[];
    ....
};
```
Directories - System View

- user view vs system view of directory tree
  - representation with “dirlists (directory files)”
- The real meaning of “A file is in a directory”
  - directory has a link to the inode of the file
- The real meaning of “A directory contains a subdirectory”
  - directory has a link to the inode of the subdirectory
- The real meaning of “A directory has a parent directory”
  - “..” entry of the directory has a link to the inode of the parent directory

Exercise

Given the following directory information:

```
$ ls -iaR home
  865 .  193 ..  277 a  520 c  491 y  492 z
home/a:
  277 .  865 ..  402 x
home/c:
  520 .  865 ..  651 d1  247 d2
home/c/d1:
  651 .  520 ..  402 xlink
home/c/d2:
  247 .  520 ..  680 xcopy
```
Exercise (cont)

a) Show the user view of this directory structure

b) Show the system view of this directory structure

c) Assume we perform the following operations:
   $ rm home/c/d2/xcopy
   $ cp home/y home/c/d1
   $ ln home/z home/c/d2/z
   $ mv home/c/d2 home/c/d1
   Show the system view of the new directory structure

Link Counts

- The kernel records the number of links to any file/directory.

- The *link count* is stored in the inode.

- The *link count* is a member of *struct stat* returned by the *stat* system call.
How to Create a New Process?

- 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 (Cont.)

- 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
How fork works?

pid_t fork(void);

- Allocates a new chunk of memory and data structures
- Copies the original process into the new process
- Adds the new process to the set of running processes
- Returns control back to both processes

Fork Implementation

```c
int main()
{
    Pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
```
Exercise

```c
main()
{
    int     ret, glob=10;
    printf("glob before fork: %d\n", glob);
    ret = fork();
    ret = vfork();
    if (ret == 0) {
        glob++;
        printf("child: glob after fork: %d\n", glob);
        exit(0);
    }
    if (ret > 0) {
        if (waitpid(ret, NULL, 0) != ret)
            printf("Wait error!\n");
        printf("parent: glob after fork: %d\n", glob);
    }
}
```

What would be the output of this program?

vfork function

```c
pid_t vfork(void);
```

- Similar to fork, but:
  - child shares all memory with parent
  - parent is suspended until the child makes an `exit` or `exec` call
### vfork example

```c
main()
{
    int ret, glob=10;

    printf("glob before fork: %d\n", glob);
    ret = vfork();

    if (ret == 0) {
        glob++;
        printf("child: glob after fork: %d\n", glob);
        exit(0);
    }

    if (ret > 0) {
        //if (waitpid(ret, NULL, 0) != ret) printf("Wait error!\n");
        printf("parent: glob after fork: %d\n", glob);
    }
}
```

### How is Environment Implemented?

- **Environment Variables**
  - `int main(int argc, char **argv, char **envp);`
  - `extern char **environ;`

- **getenv/putenv**

- **Environment list**
  - HOME=/home/stevens
  - PATH=/bin:/usr/bin
  - SHELL=/bin/sh
  - USER=stevens
  - LOGNAME=stevens

- **Environment strings**
  - HOME=/home/stevens
  - PATH=/bin:/usr/bin
  - SHELL=/bin/sh
  - USER=stevens
  - LOGNAME=stevens
Example 1

```c
#include <stdio.h>
#include <malloc.h>

extern char **environ;

main()
{
    char ** ptr;
    for (ptr=environ; *ptr != 0; ptr++)
        printf("%s\n", *ptr);
}
```

Process Accounting

- Kernel writes an accounting record each time a process terminates
- `acct struct` defined in `<sys/acct.h>`

```c
typedef u_short comp_t;
struct acct {
    char ac_flag; /* Figure 8.9 - Page 227 */
    char ac_stat; /* termination status (core flag + signal #) */
    uid_t ac_uid; gid_t ac_gid; /* real [ug]id */
    dev_t ac_tty; /* controlling terminal */
    time_t ac_btime; /* starting calendar time (seconds) */
    comp_t ac_utime; /* user CPU time (ticks) */
    comp_t ac_stime; /* system CPU time (ticks) */
    comp_t ac_cctime; /* elapsed time (ticks) */
    comp_t ac_mem; /* average memory usage */
    comp_t ac_io; /* bytes transferred (by r/w) */
    comp_t ac_rw; /* blocks read or written */
    char ac_comm[8]; /* command name: [8] for SVR4, [10] for
    4.3 BSD */
};
```
Process Accounting

- Data required for accounting record is kept in the process table
- Initialized when a new process is created
  - (e.g. after fork)
- Written into the accounting file (binary) when the process terminates
  - in the order of termination
- No records for
  - crashed processes
  - abnormal terminated processes

Pipes

- one-way data channel in the kernel
- has a reading end and a writing end

- e.g. who | sort or ps | grep ssh
Process Communication via Pipes

```c
int pipe(int filedes[2]);
```

- pipe creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading filedes[1] is for writing

Exercise

- UNIX> `sort < f1 | head -5 | cat -n`

  Hints: “`head -5`” displays first 5 lines of a file
  “`cat -n`” reads a file, writes it to stdout with line numbers

- What happens to the given process in terms of how it exits?
  - i.e. when file f1 does not exist??
Signal Disposition

- Ignore the signal (most signals can simply be ignored, except SIGKILL and SIGSTOP)
- Handle the signal disposition via a *signal handler* routine. This allows us to gracefully shutdown a program when the user presses Ctrl-C (SIGINT).
- Block the signal. In this case, the OS queues signals for possible later delivery
- Let the default apply (usually process termination)

Signals from a Process

- **int kill(pid_t pid, int sig)**
  - Can be used to send any signal to any process group or process.
    - *pid > 0*, signal *sig* is sent to *pid*.
    - *pid == 0*, *sig* is sent to every process in the process group of the current process.
    - *pid == -1*, *sig* is sent to every process except for process 1.
    - *pid < -1*, *sig* is sent to every process in the process group -*pid*.
    - *sig == 0*, no signal is sent, but error checking is performed.

*raise(signo)* causes the specified signal to be sent to the process that executes the call to raise.
Default Actions

- **Abort** – terminate the process after generating a dump
- **Exit** – terminate the process without generating a dump
- **Ignore** – the signal is ignored
- **Stop** – suspends the process
- **Continue** – resumes the process, if suspended

Receiving Signals

**Handling signals**

- Suppose kernel is returning from exception handler and is ready to pass control to process p.

- Kernel computes `pnb = pending & ~blocked`
  - The set of pending nonblocked signals for process p
- if `(pnb != 0)` {
  - Choose least nonzero bit k in `pnb` and force process p to receive signal k.
  - The receipt of the signal triggers some action by p.
  - Repeat for all nonzero k in `pnb`.
}

- Pass control to next instruction in the logical flow for p.
Masking Signals - Avoid Race Conditions

- The occurrence of a second signal while the signal handler function executes.
  - The second signal can be of different type than the one being handled, or even of the same type.
- The system also contains some features that will allow us to block signals from being processed.
  - A global context which affects all signal handlers, or a per-signal type context.

Real-time Signals

- POSIX.4 adds some additional signal facilities. The key features are:
  - The real-time signals are in addition to the existing signals, and are in the range SIGRTMIN to SIGRTMAX.
  - Real-time signals are queued, not just registered (as is done for non real-time signals).
  - The source of a real-time signal (kill, sigqueue, asynchronous I/O completion, timer expiration, etc.) is indicated when the signal is delivered.
  - A data value can be delivered with the signal.
Questions?

Hmm.

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), and J.Kim (KAIST).