CSC 4304 - Systems Programming
Fall 2008

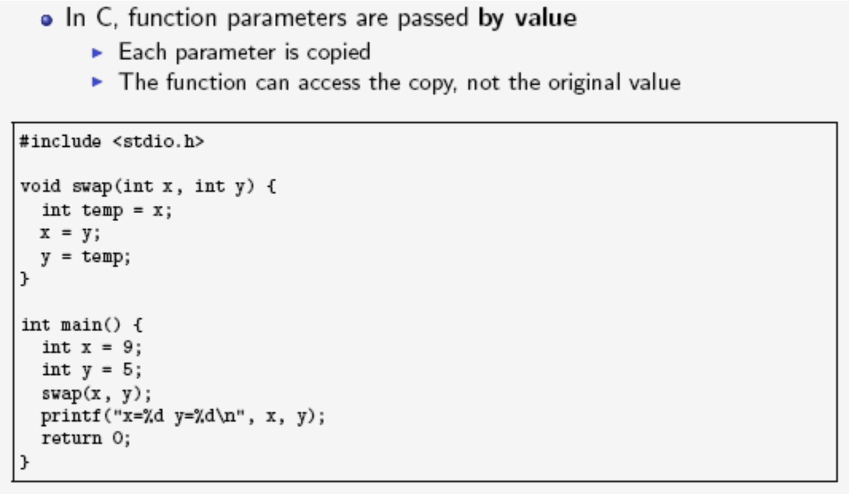
Lecture - XII
Midterm Review

Tevfik Koşar

Louisiana State University

October 14th, 2008

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

```
#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

```
#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

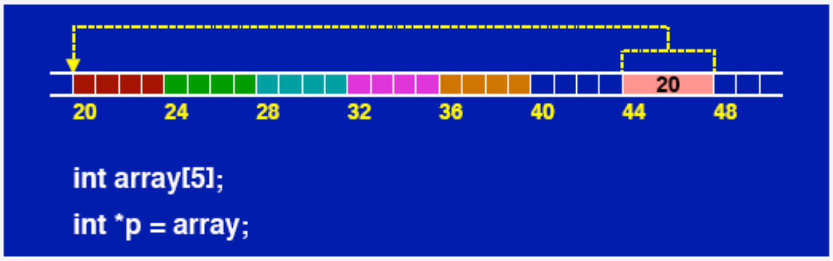
- Pointers are just a special kind of variable
- You can do **calculations** on pointers
  - You can use +, −, ++, −− on pointers
  - This has no equivalent in Java
- Be careful, operators work with the **size** of variable types!

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

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

Pointer Arithmetic

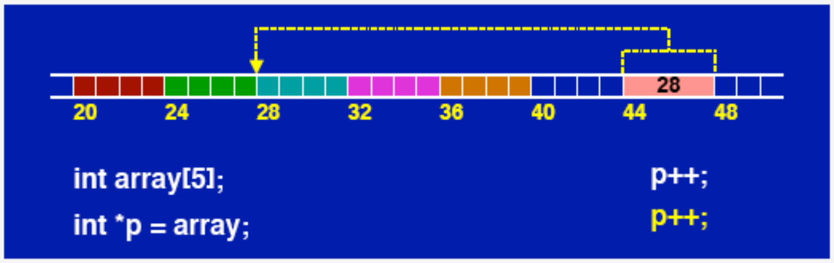
- This is obvious when using pointers as arrays:

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

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

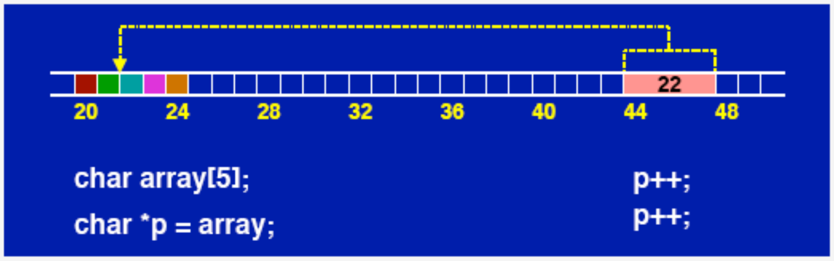


```
int array[5];
int *p = array;
```

Pointer Arithmetic



```
int array[5];                                   p++;
int *p = array;                                 p++;
```



```
char array[5];                                  p++;
char *p = array;                                p++;
```

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

```
   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

Operators	Associativity	Type
() [] -> .	left to right	primary expr.
++ *(postfix)* -- *(postfix)*	right to left	postfix
+ - ! ++ *(prefix)* -- *(prefix)* (*type*)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

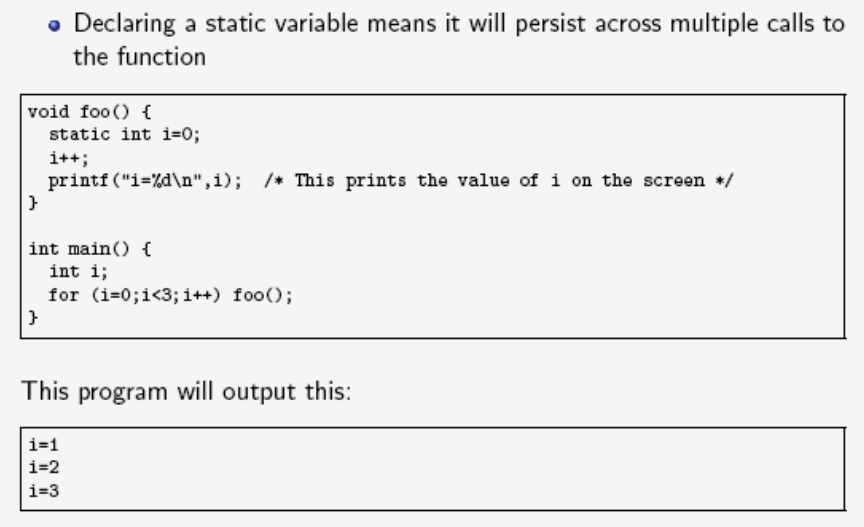
Complicated Declarations

- int *a[10]: array[10] of pointer to int
- int (*a)[10]: pointer to array[10] of int
- void *f(): function returning pointer to void
- void (*f)(): pointer to a function returning void
- char (*(*x())[])(): ??
- char (*(*x[3])())[5]: ??

Complicated Declarations

- int *a[10]: array[10] of pointer to int
- int (*a)[10]: pointer to array[10] of int
- void *f(): function returning pointer to void
- void (*f)(): pointer to a function returning void
- char (*(*x())[])(): function returning pointer to array[] of pointer to function returning char
- char (*(*x[3])())[5]: array[3] of pointer to function returning pointer to array[5] of char

Static Local Variables



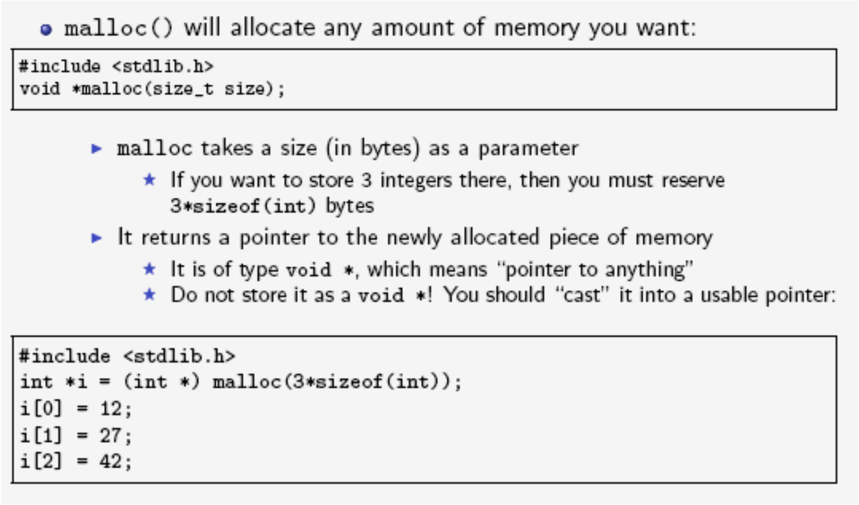
- Declaring a static variable means it will persist across multiple calls to the function

```
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:

```
#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:

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

malloc & free Example

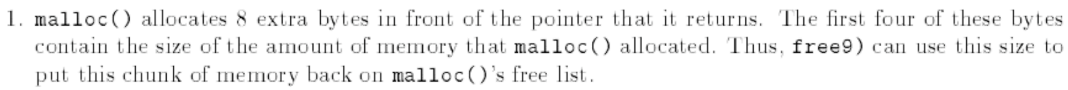
```
int main ()
{
   int x = 11;
   int *p, *q;
   p = (int *) malloc(sizeof (int));
  *p = 66;
   q = (int *) malloc(sizeof (int));
  *q = *p - 11;
   free(p);
   printf ("%d %d %d\n", x, *p, *q);
   x = 77;
   p = q;
   q = (int *) malloc(sizeof (int));
  *q = x + 11;
   printf ("%d %d %d\n", x, *p, *q);
   p = &x;
   p = (int *) malloc(sizeof (int));
  *p = 99;
   printf ("%d %d %d\n", x, *p, *q);
   q = p;
   free(q);
   printf ("%d %d %d\n", x, *p, *q);
}
```

```
./free
11 ? 55
77 55 88
77 99 88
77 ? ?
```

Exercise

The subroutine free() frees a block of memory so that it can be reused by malloc(). However, free() is passed only a pointer to the block to be freed.

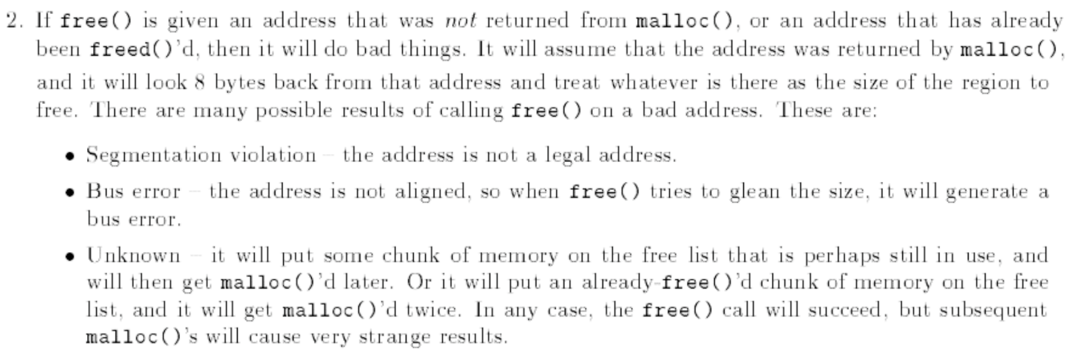
1. How does free() know how big this block is?



1. **`malloc()`** allocates 8 extra bytes in front of the pointer that it returns. The first four of these bytes contain the size of the amount of memory that **`malloc()`** allocated. Thus, **`free9)`** can use this size to put this chunk of memory back on **`malloc()`**'s free list.

Exercise *(cont.)*

2. "Bad arguments" can be passed to free(). Describe what a bad argument to free() is and what are the possible results of passing bad arguments to free()?

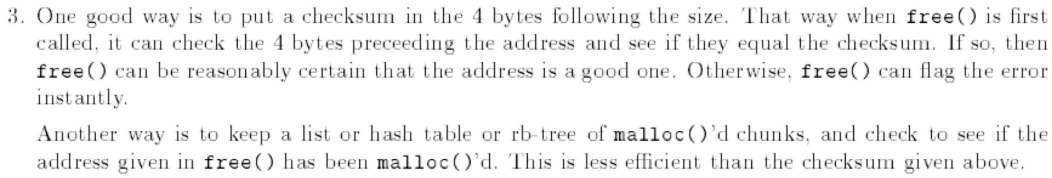


2. If **`free()`** is given an address that was *not* returned from **`malloc()`**, or an address that has already been **`freed()`**'d, then it will do bad things. It will assume that the address was returned by **`malloc()`**, and it will look 8 bytes back from that address and treat whatever is there as the size of the region to free. There are many possible results of calling **`free()`** on a bad address. These are:

   - Segmentation violation — the address is not a legal address.
   - Bus error — the address is not aligned, so when **`free()`** tries to glean the size, it will generate a bus error.
   - Unknown — it will put some chunk of memory on the free list that is perhaps still in use, and will then get **`malloc()`**'d later. Or it will put an already-**`free()`**'d chunk of memory on the free list, and it will get **`malloc()`**'d twice. In any case, the **`free()`** call will succeed, but subsequent **`malloc()`**'s will cause very strange results.

Exercise *(cont.)*

3. Free() should be fixed so that bad arguments are recognized (or are almost always recognized). Discuss how this can be achieved.



3. One good way is to put a checksum in the 4 bytes following the size. That way when `free()` is first called, it can check the 4 bytes preceeding the address and see if they equal the checksum. If so, then `free()` can be reasonably certain that the address is a good one. Otherwise, `free()` can flag the error instantly.

   Another way is to keep a list or hash table or rb-tree of `malloc()`'d chunks, and check to see if the address given in `free()` has been `malloc()`'d. This is less efficient than the checksum given above.

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, sdtout, stderr defined in <stdio.h>
    - STDIO_FILENO, STDOUT_FILENO,…

Standard I/O Eficiency

- Copy stdin to stdout using:

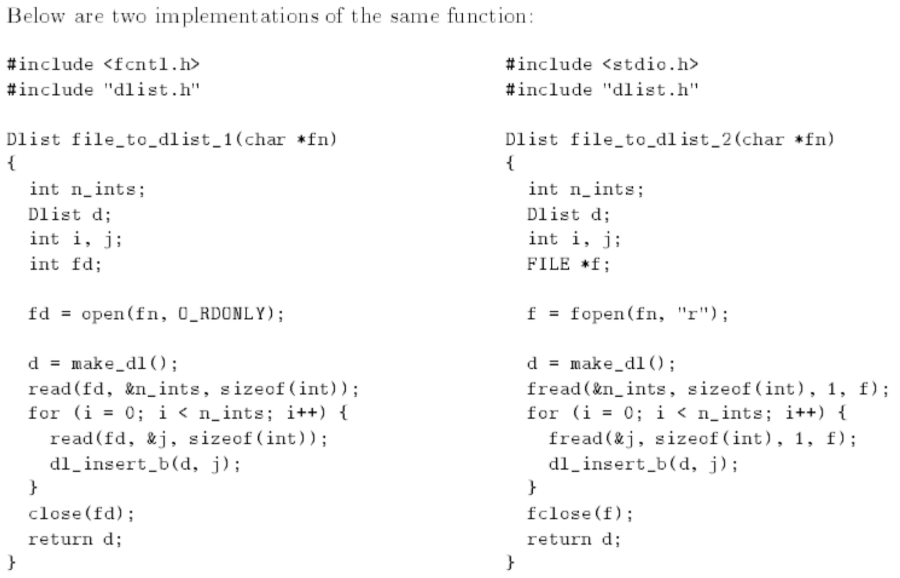
	total time	kernel time	
• fgets, fputs :	2.6 sec	| 0.3 sec	
• fgetc, fputc :	5 sec	| 0.3 sec	
• read, write :	423 sec	| 397 sec	(1 char at a time)

Effect of Buffer Size

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

buffersize	exec time
1	50.29
4	12.81
16	3.28
64	0.96
256	0.37
1024	0.22
4096	0.18
16384	0.18

Exercise



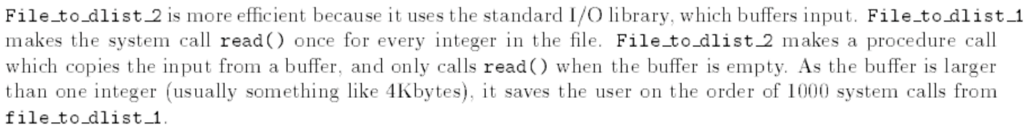
Below are two implementations of the same function:

```
#include <fcntl.h>
#include "dlist.h"

Dlist file_to_dlist_1(char *fn)
{
  int n_ints;
  Dlist d;
  int i, j;
  int fd;

  fd = open(fn, O_RDONLY);

  d = make_dl();
  read(fd, &n_ints, sizeof(int));
  for (i = 0; i < n_ints; i++) {
    read(fd, &j, sizeof(int));
    dl_insert_b(d, j);
  }
  close(fd);
  return d;
}
```

```
#include <stdio.h>
#include "dlist.h"

Dlist file_to_dlist_2(char *fn)
{
  int n_ints;
  Dlist d;
  int i, j;
  FILE *f;

  f = fopen(fn, "r");

  d = make_dl();
  fread(&n_ints, sizeof(int), 1, f);
  for (i = 0; i < n_ints; i++) {
    fread(&j, sizeof(int), 1, f);
    dl_insert_b(d, j);
  }
  fclose(f);
  return d;
}
```

Which of the functions above will be more efficient if the file being read is large, and why?

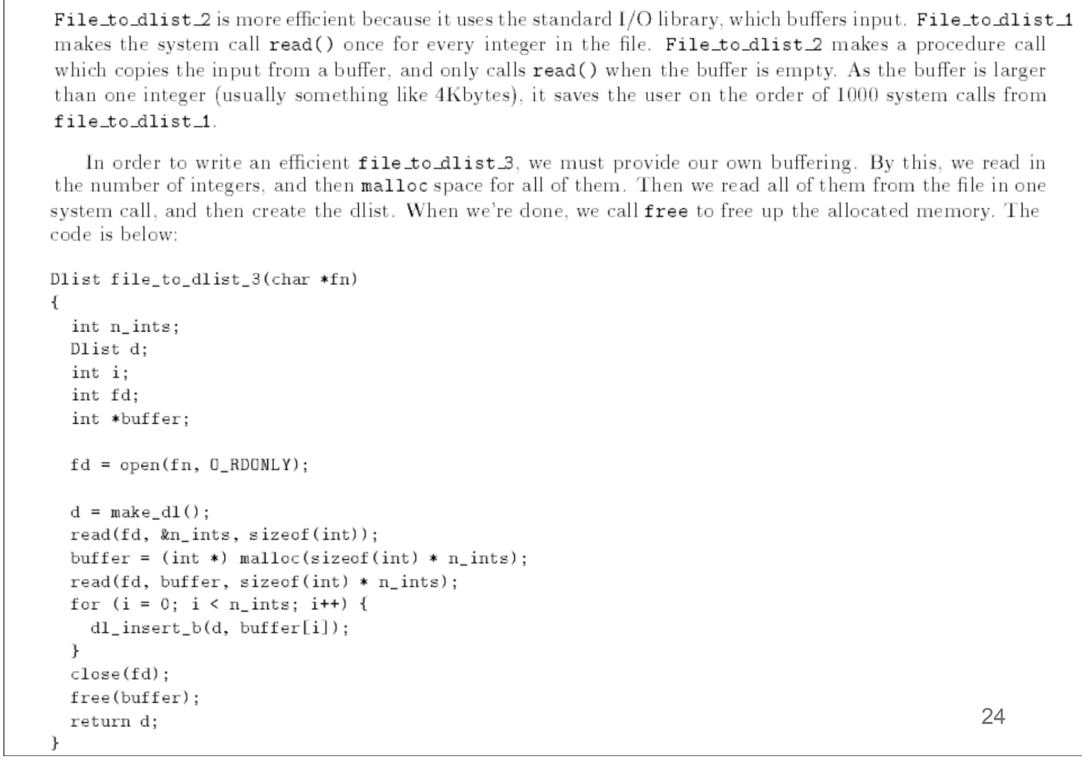
Solution



File_to_dlist_2 is more efficient because it uses the standard I/O library, which buffers input. File_to_dlist_1 makes the system call read() once for every integer in the file. File_to_dlist_2 makes a procedure call which copies the input from a buffer, and only calls read() when the buffer is empty. As the buffer is larger than one integer (usually something like 4Kbytes), it saves the user on the order of 1000 system calls from file_to_dlist_1.

Write a function file_to_dlist_3() which is functionally equivalent to these two functions, and is at least as efficient as the best of these two.

Solution



File_to_dlist_2 is more efficient because it uses the standard I/O library, which buffers input. File_to_dlist_1 makes the system call read() once for every integer in the file. File_to_dlist_2 makes a procedure call which copies the input from a buffer, and only calls read() when the buffer is empty. As the buffer is larger than one integer (usually something like 4Kbytes), it saves the user on the order of 1000 system calls from file_to_dlist_1.

In order to write an efficient file_to_dlist_3, we must provide our own buffering. By this, we read in the number of integers, and then malloc space for all of them. Then we read all of them from the file in one system call, and then create the dlist. When we're done, we call free to free up the allocated memory. The code is below:

```
Dlist file_to_dlist_3(char *fn)
{
  int n_ints;
  Dlist d;
  int i;
  int fd;
  int *buffer;

  fd = open(fn, O_RDONLY);

  d = make_dl();
  read(fd, &n_ints, sizeof(int));
  buffer = (int *) malloc(sizeof(int) * n_ints);
  read(fd, buffer, sizeof(int) * n_ints);
  for (i = 0; i < n_ints; i++) {
    dl_insert_b(d, buffer[i]);
  }
  close(fd);
  free(buffer);
  return d;
}
```

Restrictions

Type	r	w	a	r+	w+	a+
File exists?	Y			Y		
Truncate		Y			Y	
R	Y			Y	Y	Y
W		Y	Y	Y	Y	Y
W only at end			Y			Y

* 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.

```
struct stat {
    mode_t   st_mode; /* type & mode */
    ino_t    st_ino; /* i-node number */
    dev_t    st_dev; /* device no (filesystem) */
    dev_t    st_rdev; /* device no for special file */
    nlink_t  st_nlink; /* # of links */
    uid_t    st_uid;        gid_t    st_gid;
    off_t    st_size; /* sizes in byes */
    time_t   st_atime; /* last access time */
    time_t   st_mtime; /* last modification time */
    time_t   st_ctime; /* time for last status change */
    long     st_blk_size; /* best I/O block size */
    long     st_blocks; /* number of 512-byte blocks allocated */
  };
```

Directories

- dirent : file system independent directory entry

```
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

Example inode listing

```
$ ls -iaR demodir
865 .       193 ..      277 a       520 c     491 y

demodir/a:
277 .       865 ..      402 x

demodir/c:
520 .      865 ..      651 d1          247 d2

demodir/c/d1:
651 .      520 ..      402 xlink

demodir/c/d2:
247 .      520 ..      680 xcopy
```

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.

Set-User-ID Bit

- How can a regular user change his/her password?

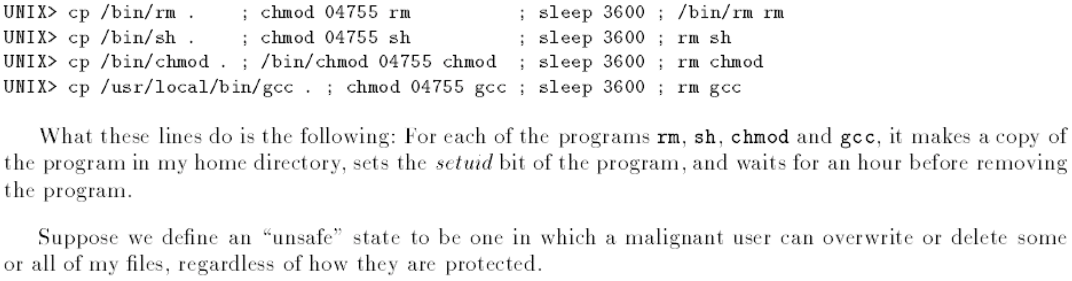
```
cs4304_kos@classes:~> ls -l /etc/passwd
-rw-r--r--  1 root root 70567 2008-09-23 09:28 /etc/passwd
```

- Permission is given to the program, not to you!

```
cs4304_kos@classes:~> ls -l /usr/bin/passwd
-rwsr-xr-x  1 root shadow 79520 2005-09-09 15:56 /usr/bin/passwd
```

```
04000 : set user ID
02000 : set group ID
01000 : sticky bit - keep the program in swap device
```

Exercise



```
UNIX> cp /bin/rm .    ; chmod 04755 rm          ; sleep 3600 ; /bin/rm rm
UNIX> cp /bin/sh .    ; chmod 04755 sh          ; sleep 3600 ; rm sh
UNIX> cp /bin/chmod . ; /bin/chmod 04755 chmod  ; sleep 3600 ; rm chmod
UNIX> cp /usr/local/bin/gcc . ; chmod 04755 gcc ; sleep 3600 ; rm gcc
```

What these lines do is the following: For each of the programs **rm**, **sh**, **chmod** and **gcc**, it makes a copy of the program in my home directory, sets the *setuid* bit of the program, and waits for an hour before removing the program.

Suppose we define an "unsafe" state to be one in which a malignant user can overwrite or delete some or all of my files, regardless of how they are protected.

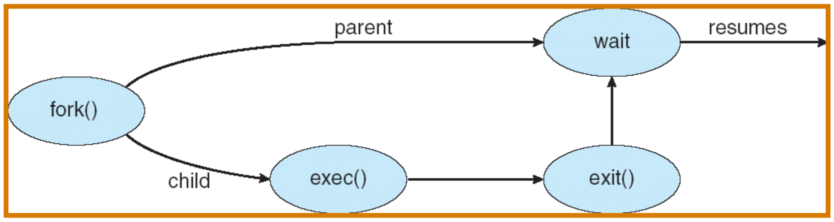
Which lines are unsafe? Why?

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

```
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 */
```

vfork function

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

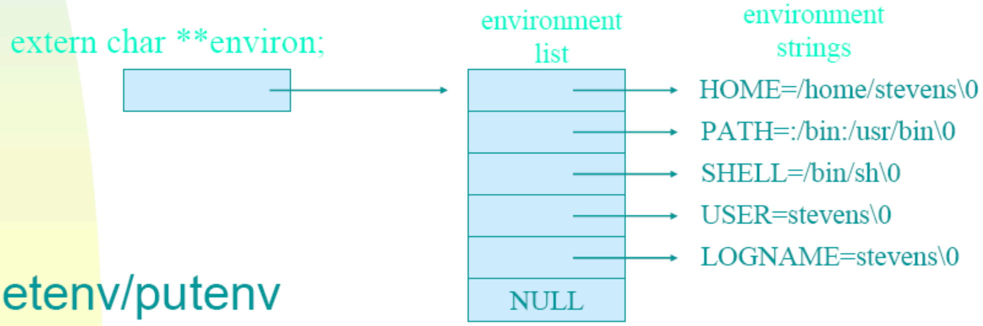
- Similar to fork, but:
  - child shares all memory with parent
  - parent is suspended until the child makes an **exit** or **exec** call

vfork example

```
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 agrc, char **argv, char **envp);



extern char **environ;

environment list

environment strings

HOME=/home/stevens\0

PATH=:/bin:/usr/bin\0

SHELL=/bin/sh\0

USER=stevens\0

LOGNAME=stevens\0

NULL

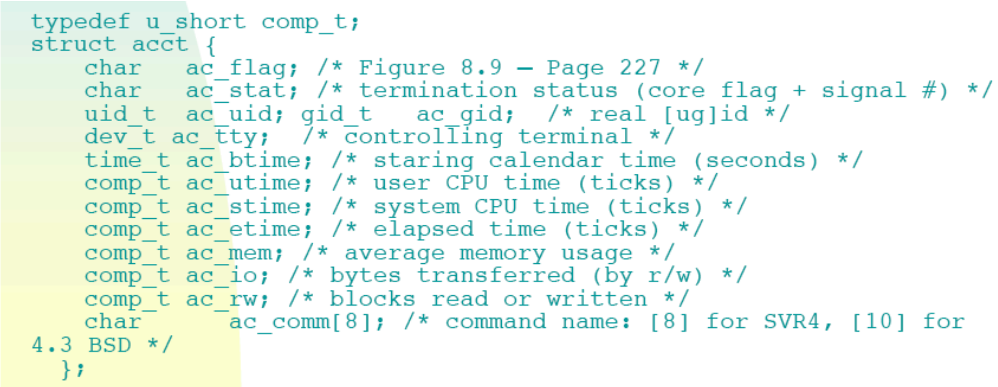
- getenv/putenv

Example 1

```
#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>



```
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; /* staring calendar time (seconds) */
    comp_t ac_utime; /* user CPU time (ticks) */
    comp_t ac_stime; /* system CPU time (ticks) */
    comp_t ac_etime; /* 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

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.

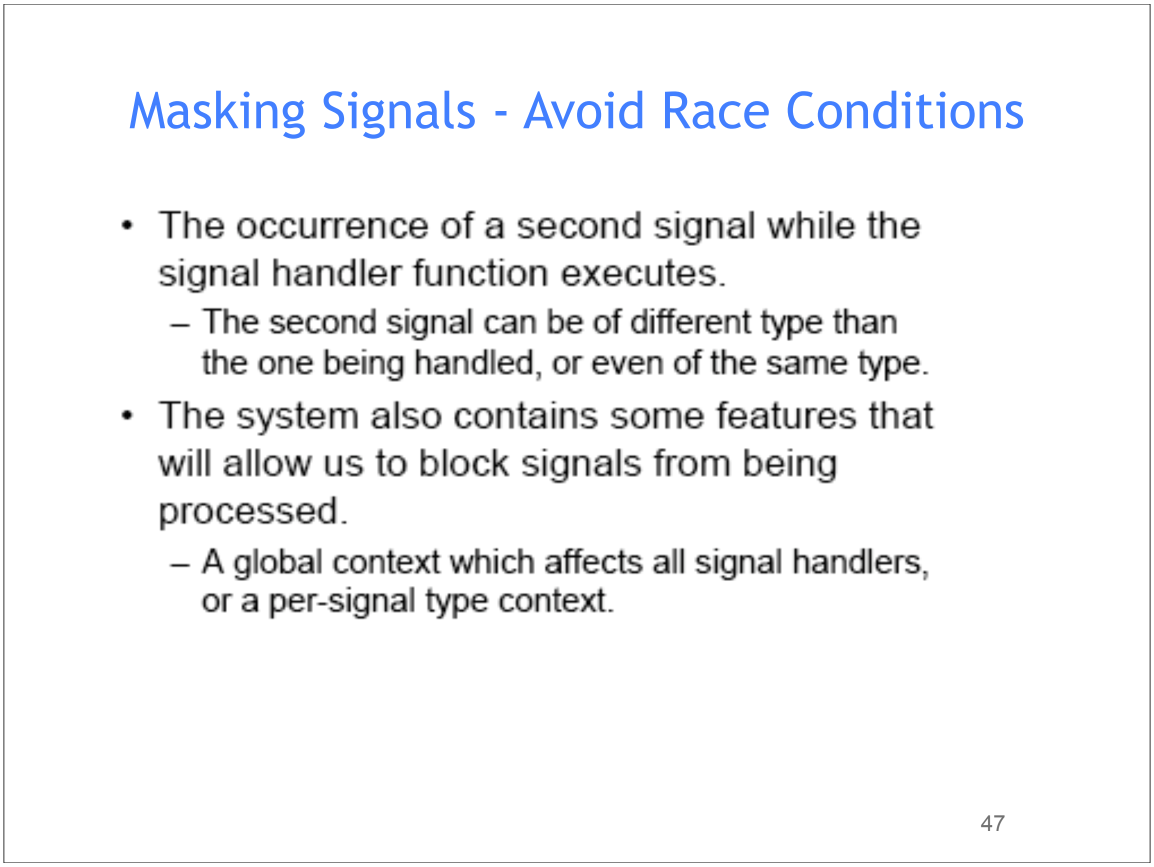
Default Actions

- 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

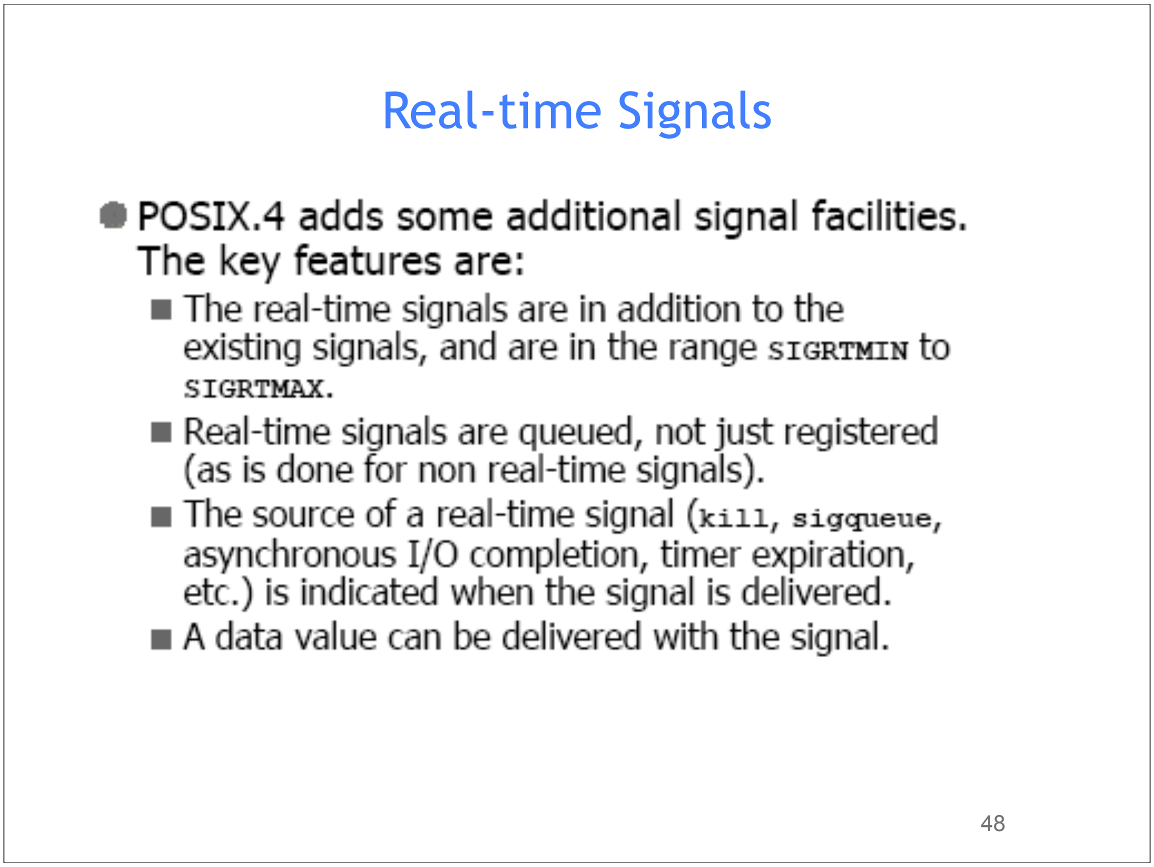
- **Handling signals**
  - Suppose kernel is returning from exception handler and is ready to pass control to process p.
  - Kernel computes **pnb** = **pending** & ~**blocked**
    - The set of pending nonblocked signals for process p
  - if (**pnb** != 0) {
    - Choose least nonzero bit k in **pnb** 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.

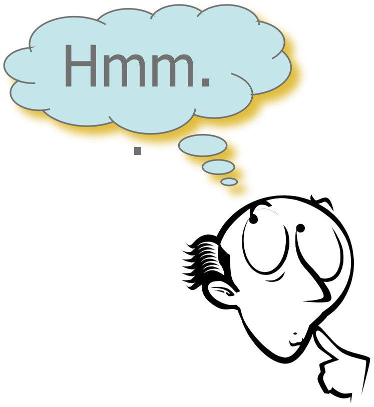
Exercise

Write a program which blocks all of the other signals (except the ones which cannot be blocked) if a signal arrives (same one or different one) while your program is inside a signal handler. Hint use sigaction() function.

Solution

```
   main()
{
        struct sigaction newhandler;         /* new settings        */
        sigset_t         blocked;            /* set of blocked sigs */
        void             inthandler();       /* the handler         */

        newhandler.sa_handler = inthandler;  /* handler function    */
        sigfillset(&blocked);               /* mask all signals      */
        newhandler.sa_mask = blocked;       /* store blockmask     */

        int i;
        for (i=1; i<65;i++)
                if (i!=9 && i!=19 && i!=32 && i!=33)
            /* catch all except these signals */

                if ( sigaction(i, &newhandler, NULL) == -1 )
                          printf("error with signal %d\n", i);
        while(1){}
}
```

Questions?



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).