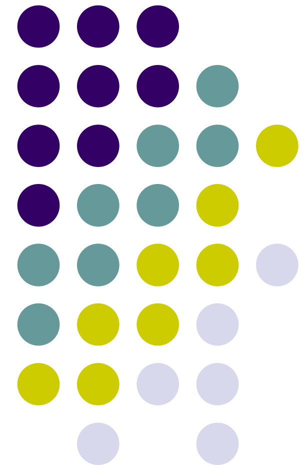
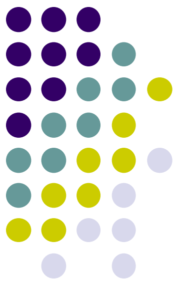
Unix System Calls

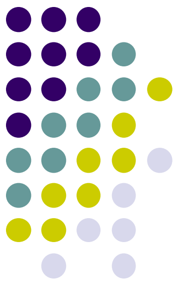
Gwan-Hwan Hwang
Dept. CSIE
National Taiwan Normal University
2019.12.23



UNIX System Overview

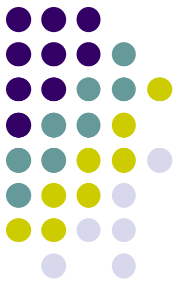


- UNIX Architecture
- Login Name
- Shells
- Files and Directories
  - File System
  - Filename
  - Pathname
  - Working Directory, Home Directory

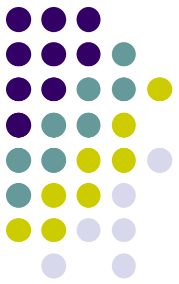


- Advanced UNIX Programming, the second edition, by Marc J. Rochkind
- Unix System Calls
  - http://www.di.uevora.pt/~lmr/syscalls.html

System call error



- When a system call discovers and error, it returns -1 and stores the reason the called failed in an external variable named "errno".
- The "/usr/include/errno.h" file maps these error numbers to manifest constants, and it these constants that you should use in your programs.



user application

open ( )

user mode

system call interface

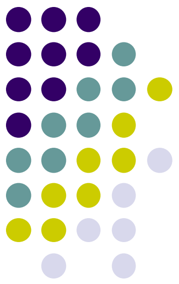
kernel mode

i

open ( )

Implementation
of open ( )
system call

return



```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return o;
}
```

user mode

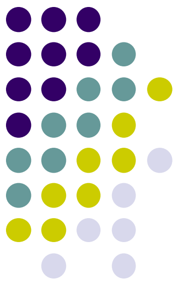
standard C library

kernel mode

write ( )

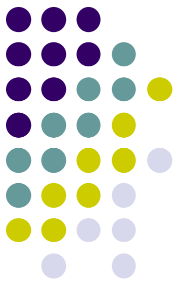
write ( ) system call

System call error (Cont’d)

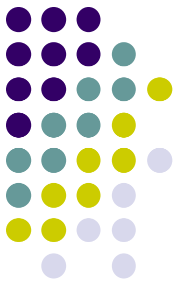


- When a system call returns successfully, it returns something other than -1, but it does not clear "errno". "errno" only has meaning directly after a system call that returns an error.

File structure related system calls



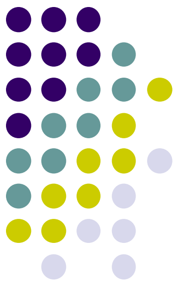
- creat()
- open()
- close()
- read()
- write()
- lseek()
- dup()
- link()
- unlink()
- stat()
- fstat()
- access()
- chmod()
- chown()
- umask()
- ioctl()



File Structure Related System Calls

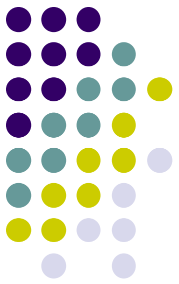
- The file structure related system calls available in the UNIX system let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices.

File Structure Related System Calls (Cont’d)



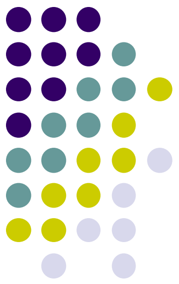
- To a process then, all input and output operations are synchronous and unbuffered.
- All input and output operations start by opening a file using either the "creat()" or "open()" system calls.
  - These calls return a file descriptor that identifies the I/O channel.

File descriptors



- Each UNIX process has 20 file descriptors at it disposal, numbered 0 through 19.
- The first three are already opened when the process begins
  - 0: The standard input
  - 1: The standard output
  - 2: The standard error output
- When the parent process forks a process, the child process inherits the file descriptors of the parent.

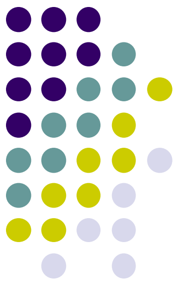
creat() system call



- The prototype for the creat() system call is:

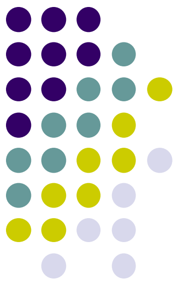
```
int creat(file_name, mode)
char *file_name;
int mode;
```

creat() system call (Cont'd)



- The mode is usually specified as an octal number such as 0666 that would mean read/write permission for owner, group, and others or the mode may also be entered using manifest constants defined in the "/usr/include/sys/stat.h" file.

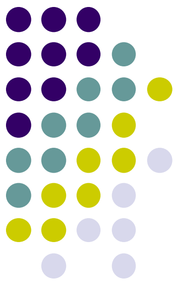
creat() system call (Cont'd)



- The following is a sample of the manifest constants for the mode argument as defined in /usr/include/sys/stat.h:

```
#define S_IRWXU 0000700    /* -rwx------ */
#define S_IREAD 0000400    /* read permission, owner */
#define S_IRUSR S_IREAD
#define S_IWRITE 0000200   /* write permission, owner */
#define S_IWUSR S_IWRITE
#define S_IEXEC 0000100    /* execute/search permission, owner */
#define S_IXUSR S_IEXEC
#define S_IRWXG 0000070    /* ----rwx--- */
#define S_IRGRP 0000040    /* read permission, group */
#define S_IWGRP 0000020    /* write    "      "   */
#define S_IXGRP 0000010    /* execute/search "   "   */
#define S_IRWXO 0000007    /* -------rwx */
#define S_IROTH 0000004    /* read permission, other */
#define S_IWOTH 0000002    /* write    "      "   */
#define S_IXOTH 0000001    /* execute/search "   "   */
```

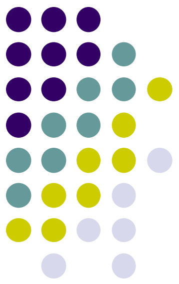
open() system call



- The prototype for the open() system call is:

```
#include <fcntl.h>
int open(file_name, option_flags [, mode])
char *file_name;
int option_flags, mode;
```

open() system call (Cont'd)

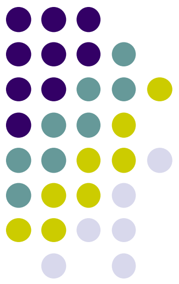


- The allowable option_flags as defined in "/usr/include/fcntl.h" are:

```
#define  O_RDONLY 0        /* Open the file for reading only */
#define  O_WRONLY 1        /* Open the file for writing only */
#define  O_RDWR   2     /* Open the file for both reading and writing*/
#define  O_NDELAY 04       /* Non-blocking I/O */
#define  O_APPEND 010      /* append (writes guaranteed at the end) */
#define  O_CREAT 00400  /*open with file create (uses third open arg) */
#define  O_TRUNC  01000    /* open with truncation */
#define  O_EXCL   02000    /* exclusive open */
```

- Multiple values are combined using the | operator (i.e. bitwise OR).

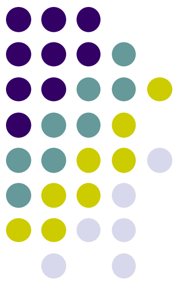
close() system call



- To close a channel, use the close() system call. The prototype for the close() system call is:

```
int close(file_descriptor)
int file_descriptor;
```

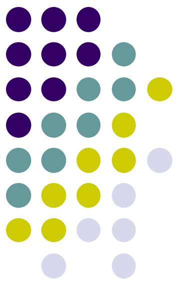
read() & write() system calls



- The read() system call does all input and the write() system call does all output.

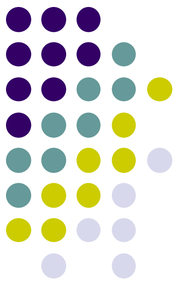
```
int read(file_descriptor, buffer_pointer, transfer_size)
int file_descriptor;
char *buffer_pointer;
unsigned transfer_size;

int write(file_descriptor, buffer_pointer, transfer_size)
int file_descriptor;
char *buffer_pointer;
unsigned transfer_size;
```

lseek() system call



- The UNIX system file system treats an ordinary file as a sequence of bytes.
- Generally, a file is read or written sequentially -- that is, from beginning to the end of the file. Sometimes sequential reading and writing is not appropriate.
- Random access I/O is achieved by changing the value of this file pointer using the lseek() system call.

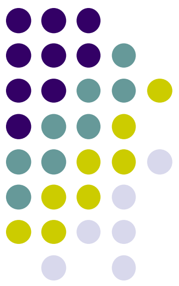
lseek() system call (Cont'd)



```
long lseek(file_descriptor, offset, whence)
int file_descriptor;
long offset;
int whence;
```

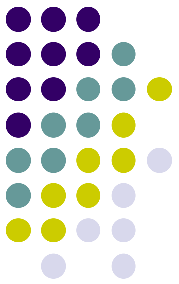
whence	new position
0	offset bytes into the file
1	current position in the file plus offset
2	current end-of-file position plus offset

dup() system call



- The dup() system call duplicates an open file descriptor and returns the new file descriptor.
- The new file descriptor has the following properties in common with the original file descriptor:
  - refers to the same open file or pipe.
  - has the same file pointer -- that is, both file descriptors share one file pointer.
  - has the same access mode, whether read, write, or read and write.

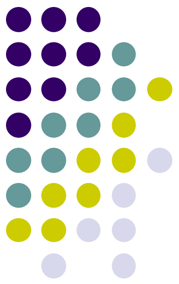
dup() system call (Cont’d)



- dup() is guaranteed to return a file descriptor with the lowest integer value available. It is because of this feature of returning the lowest unused file descriptor available that processes accomplish I/O redirection.

```
int dup(file_descriptor)
int file_descriptor;
```

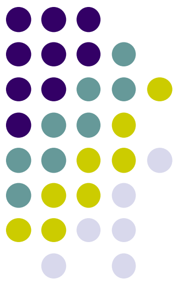
link() system call



- The UNIX system file structure allows more than one named reference to a given file, a feature called "aliasing".
- Making an alias to a file means that the file has more than one name, but all names of the file refer to the same data.

```
int link(original_name, alias_name)
char *original_name, *alias_name;
```

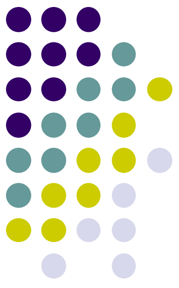
unlink() system call



- The opposite of the link() system call is the unlink() system call.
- The prototype for unlink() is:

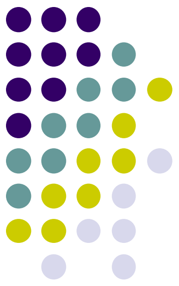
```
 int unlink(file_name)
char *file_name;
```

Process Related System Calls



- The UNIX system provides several system calls to
  - create and end program,
  - to send and receive software interrupts,
  - to allocate memory, and to do other useful jobs for a process.
- Four system calls are provided for creating a process, ending a process, and waiting for a process to complete.
  - These system calls are fork(), the "exec" family, wait(), and exit().

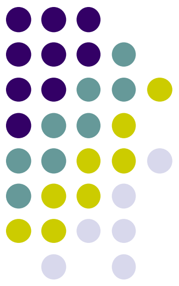
exec() system calls



- The UNIX system calls that transform a executable binary file into a process are the "exec" family of system calls. The prototypes for these calls are:

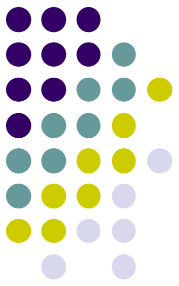
```
int execl(file_name, arg0 [, arg1, ..., argn], NULL)
char *file_name, *arg0, *arg1, ..., *argn;

int execv(file_name, argv)
char *file_name, *argv[];

int execle(file_name, arg0 [, arg1, ..., argn], NULL, envp)
char *file_name, *arg0, *arg1, ..., *argn, *envp[];

int execve(file_name, argv, envp)
char *file_name, *argv[], *envp[];

int execlp(file_name, arg0 [, arg1, ..., argn], NULL)
char *file_name, *arg0, *arg1, ..., *argn;

int execvp(file_name, argv)
char *file_name, *argv[];
```

exec() system calls (Cont'd)



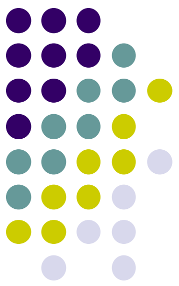
- Unlike the other system calls and subroutines, a successful exec system call does not return. Instead, control is given to the executable binary file named as the first argument.
- When that file is made into a process, that process replaces the process that executed the exec system call -- a new process is not created.

exec() system calls (Cont'd)



- Letters added to the end of exec indicate the type of arguments:
  - l argn is specified as a list of arguments.
  - v argv is specified as a vector (array of character pointers).
  - e environment is specified as an array of character pointers.
  - p user's PATH is searched for command, and command can be a shell program

fork() system call



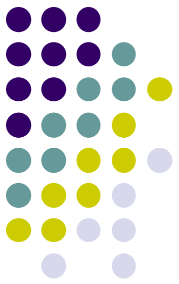
- To create a new process, you must use the fork() system call.
  - The prototype for

the fork() system call is:

int fork()

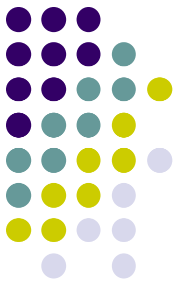
- fork() causes the UNIX system to create a new process, called the "child process", with a new process ID. The *contents* of the child process are identical to the *contents* of the parent process.

fork() system call (Cont’d)



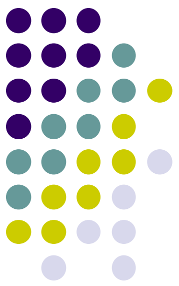
- The new process inherits several characteristics of the old process. Among the characteristics inherited are:
  - The environment.
  - All signal settings.
  - The set user ID and set group ID status.
  - The time left until an alarm clock signal.
  - The current working directory and the root directory.
  - The file creation mask as established with umask().
- fork() returns zero in the child process and non-zero (the child's process ID) in the parent process.

wait() system call



- You can control the execution of child processes by calling wait() in the parent.
- wait() forces the parent to suspend execution until the child is finished.
- wait() returns the process ID of a child process that finished.
- If the child finishes before the parent gets around to calling wait(), then when wait() is called by the parent, it will return immediately with the child's process ID.

wait() system call (Cont’d)

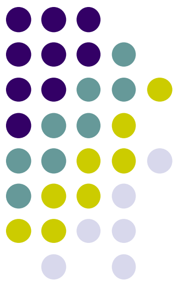


- The prototype for the wait() system call is:

```
 int wait(status)
int *status;
```

- “status” is a pointer to an integer where the UNIX system stores the value returned by the child process. wait() returns the process ID of the process that ended.

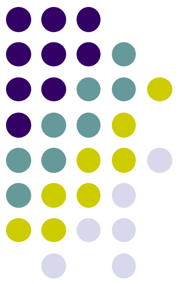
exit() system call



- The exit() system call ends a process and returns a value to it parent.
- The prototype for the exit() system call is:

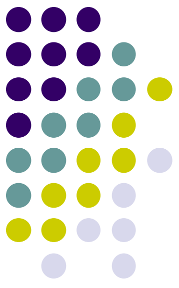
```
void exit(status)
int status;
```

- where status is an integer between 0 and 255. This number is returned to the parent via wait() as the exit status of the process.
- By convention, when a process exits with a status of zero that means it didn't encounter any problems; when a process exit with a non-zero status that means it did have problems.



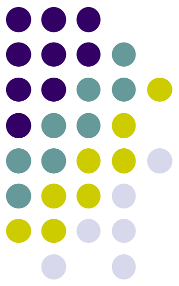
- Following are some example programs that demonstrate the use of fork(), exec(), wait(), and exit():
  - status.c
    - status>>8，講義程式有誤
  - myshell.c
  - newdir.c

Software Interrupt



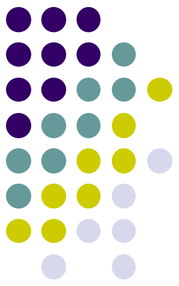
- The UNIX system provides a facility for sending and receiving software interrupts, also called SIGNALS.
- Signals are sent to a process when a predefined condition happens.
- The number of signals available is system dependent.
- The signal name is defined in /usr/include/sys/signal.h as a manifest constant.

Signal



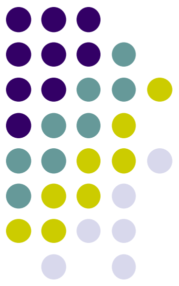
- Programs can respond to signals three different ways.
  - **Ignore the signal.** This means that the program will never be informed of the signal no matter how many times it occurs. The only exception to this is the SIGKILL signal which can neither be ignored nor caught.
  - **A signal can be set to its default state**, which means that the process will be ended when it receives that signal. In addition, if the process receives any of SIGQUIT, SIGILL, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, or SIGSYS, the UNIX system will produce a core image (core dump), if possible, in the directory where the process was executing when it received the program-ending signal.
  - **Catch the signal**. When the signal occurs, the UNIX system will transfer control to a previously defined subroutine where it can respond to the signal as is appropriate for the program.

Signal



- Related system calls
  - signal
  - kill
  - alarm

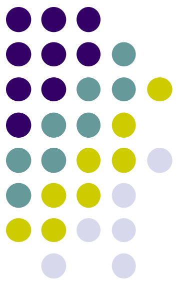
signal() system call



- You define how you want to respond to a signal with the signal() system call. The prototype is:

```
#include <sys/signal.h>

int (* signal ( signal_name, function ))
int signal_name;
int (* function)();
```

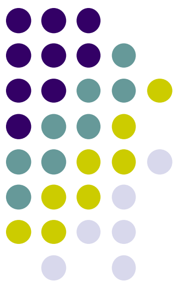
kill() system call



- The UNIX system sends a signal to a process when something happens, such as typing the interrupt key on a terminal, or attempting to execute an illegal instruction. Signals are also sent to a process with the kill() system call. Its prototype is:

```
int kill (process_id, signal_name )
int process_it, signal_name;
```

alarm() system call

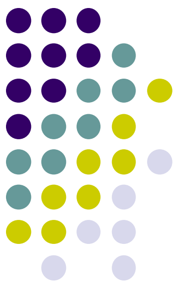


- Every process has an alarm clock stored in its system-data segment. When the alarm goes off, signal SIGALRM is sent to the calling process. A child inherits its parent's alarm clock value, but the actual clock isn't shared.
- The alarm clock remains set across an exec. The prototype for alarm() is:

  unsigned int alarm(seconds)

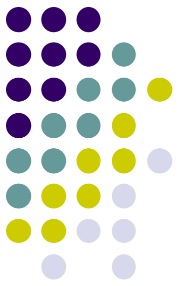
  unsigned int seconds;
- Check
  - timesup.c

vfork() system all



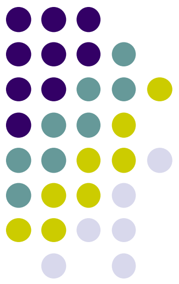
- vfork() differs from fork(2) in that the parent is suspended until the child terminates (either normally, by calling _exit(2), or abnormally, after delivery of a fatal signal), or it makes a call to execve(2). Until that point, the child shares all memory with its parent, including the stack.

pty: pseudoterminal interfaces



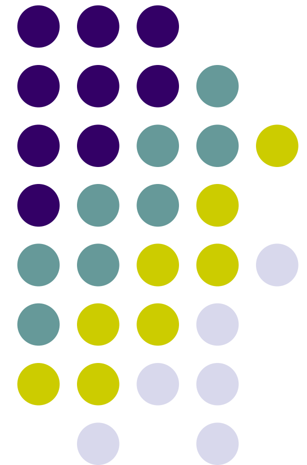
- man 7 pty
- A pseudoterminal, pseudotty, or PTY is a pair of pseudo-devices, one of which, the slave, emulates a hardware text terminal device, the other of which, the master, provides the means by which a terminal emulator process controls the slave.

forkpty()

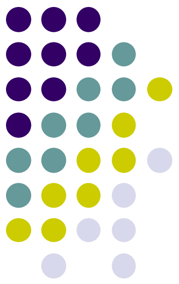


- Create a new process attached to an available pseudo-terminal
- Each subsequent read() from the master side will return data written on the slave part of the pseudo terminal.

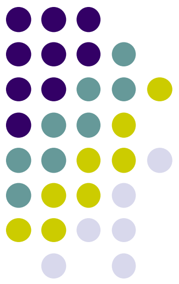
Basic Interprocess Communication



Pipes

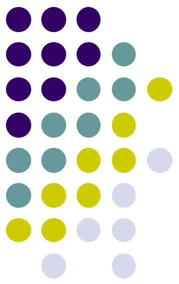


- Pipes are familiar to most UNIX users as a shell facility
  - who | sort | pr
- Related system calls
  - pipe
  - dup



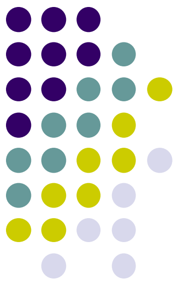
```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
  int fd;

  fd = open("foo.bar",O_WRONLY | O_CREAT, S_IREAD | S_IWRITE );
  if (fd == -1)
    {
    perror("foo.bar");
    exit (1);
    }
  close(1);        /* close standard output  */
  dup(fd);      /* fd will be duplicated into standard out's slot */
  close(fd);       /* close the extra slot */
  printf("Hello, world!\n");    /* should go to file foo.bar */
  exit (0);        /* exit() will close the files */
}
```

Interprocess Communication



- UNIX System V allows processes to communicate with one another using
  - pipes,
  - messages,
  - semaphores,
  - and shared memory.
- This section describes how to communicate using pipes.

pipe() system call



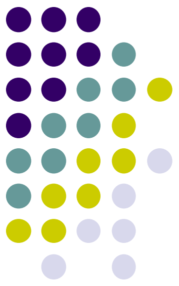
- The prototype for pipe() is:

  int pipe (file_descriptors)

  int file_descriptors[2];

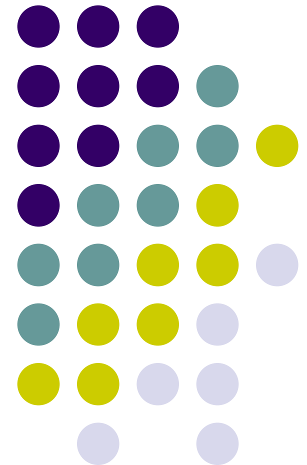
  - where file_descriptors[2] is an array that pipe() fills with a file
    - descriptor opened for reading, file_descriptor[0],
    - opened for writing, file_descriptor[1].

pipe() system call (Cont'd)

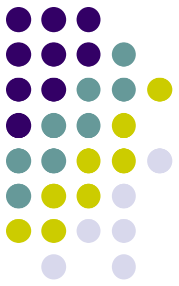


- Related system calls
  - Read, write, close, fcntl
- Check who_wc.c
  - It demonstrates a one-way pipe between two processes.
  - This program implements the equivalent of the shell command: who | wc -l
  - which will count the number of users logged in.

Advanced interprocess communication

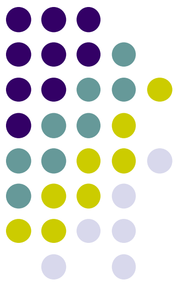


Message system calls (SYSTEM V)



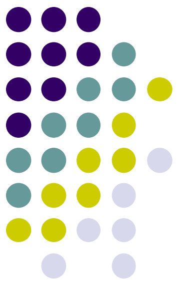
- Related system calls
  - msgget
  - msgsnd
  - msgrcv
  - msgctl
- To use message you start with msgget, which is analogous to open. It takes a key, which must be a long integer, and returns an integer called the queue-ID.
- To check the queue:
  - ipcs, ipcrm msg 0

sender.c



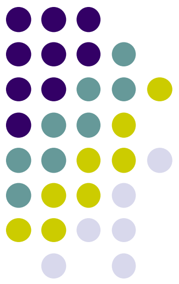
```
#include <sys/ipc.h>
#include <sys/msg.h>

main()
{

    int msqid;
    char *buf="I enjoy the OS course very much.\n";

    msqid = msgget(0x888, IPC_CREAT|0660);

    printf("To send %d bytes\n",strlen(buf));

    msgsnd(msqid, buf, strlen(buf), 0);  /* stick him on the queue */
    printf("The sender has successfully sent the message\n");
}
```

receiver.c

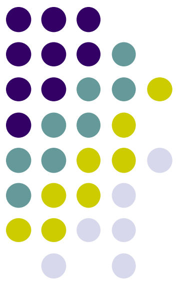


```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

main()
{

    key_t key;
    int msqid;
    char buf[255];

    key = ftok("/home/beej/somefile", 'b');
    msqid = msgget(0x888, IPC_CREAT|0660);

    msgrcv(msqid, &buf,255, 0, 0);
    printf("The receiver has successfully received the message.\n");
    printf("The message is => %s\n", buf);
}
```

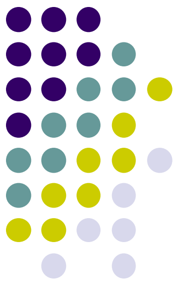
Shared memory



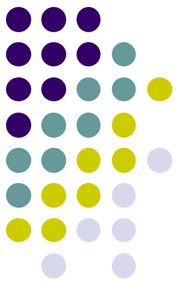
- Related system calls
  - shmget
  - shmat
  - shmdt
  - shmctl
- The shared memory is called a segment.
- A segment is first created outside the address space of any process, and then each process that wants to access it executes a system call to map it into its own address space.
- Subsequent access to the shared memory is via normal instructions that store and fetch data.



```
#define PERMS 0666

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmid;
char *mesgptr;

main()
{
  shmid = shmget(SHMKEY,1000,PERMS|IPC_CREAT);
  mesgptr = (char *)shmat(shmid,(char *)0,0);
  strcpy(mesgptr,"test share memory");
  shmdt(mesgptr);
}
```



```
#define PERMS 0666

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmid;
char *mesgptr;

main()
{
   shmid = shmget(SHMKEY,1000,0);
   mesgptr = (char *)shmat(shmid,(char *)0,0);
   printf("%s\n",mesgptr);
   shmdt(mesgptr);
}
```

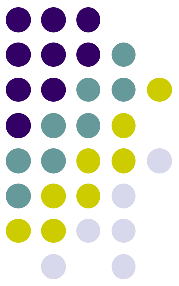


Setuid programs

- Setuid, which stands for set user ID on execution, is a special type of file permission in Unix and Unix-like operating systems such as Linux and BSD. It is a security tool that permits users to run certain programs with escalated privileges.
- When an executable file's setuid permission is set, users may execute that program with a level of access that matches the user who owns the file. For instance, when a user wants to change their password, they run the passwd command. The passwd program is owned by the root account and marked as setuid, so the user is temporarily granted root access for that very limited purpose.

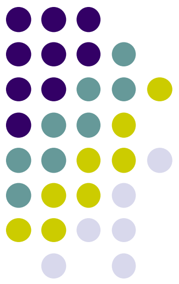
- Setting the setuid permission of a file
  - chmod u+s myfile
  - chmod g+s myfile2

Setting the setuid permission of a file



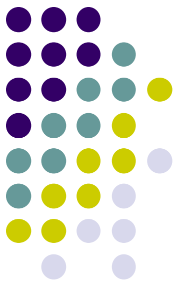
- chmod u+s myfile
- chmod g+s myfile2

File Status



- The i-node data structure holds all the information about a file except the file's name and its contents.
- Sometimes your programs need to use the information in the i-node structure to do some job. You can access this information with the stat() and fstat() system calls.

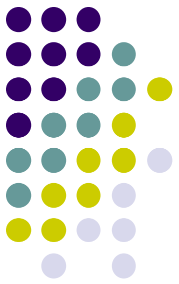
stat() and fstat() system calls



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

int stat(file_name, stat_buf)
char *file_name;
struct stat *stat_buf;

int fstat(file_descriptor, stat_buf)
int file_descriptor;
struct stat *stat_buf;
```

- Check stat.c

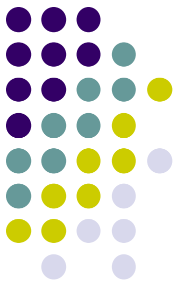
access() system call



- To determine if a file is accessible to a program, the access() system call may be used.
- The prototype for the access() system call is:

```
int access(file_name, access_mode)
char *file_name;
int access_mode;
```

Miscellaneous System Calls / Examples



- Directories
  - A directory is simply a special file that contains (among other information) i-number/filename pairs
- System V Directories
  - A directory contains structures of type direct, defined in the include file /usr/include/sys/dir.h. The include file /usr/include/sys/types.h must also be included to define the types used by the structure. The directory structure is:

```
#define DIRSIZ   14

struct direct  {
      ino_t    d_ino;
      char     d_name[DIRSIZ];
};
```

- Check my_ls.c