Process (or *task*) - informally is a running program. Some characteristics of a process are:

- one or more threads (in the "old" days a UNIX process could run only one thread)
- single virtual memory accessible to those threads with same access rights
- other non-memory (files on disk, network services, etc.). Two general approaches for process access rights:
  i. process can hold specific *capability* (e.g., writing to a particular file)
  ii. process can hold a general *credential* (e.g., identification of the user running the process). Credential mechanisms include *access control lists*
- resource allocation context - limited resources (e.g., memory, disk space, CPU time, ...) are associated with a process for two reasons: (1) they can be released/reallocated on termination, and (2) quota or billing
- miscellaneous context - other process state, e.g., current working directory

1. Operations on Processes

A process (the *parent* process) can typically create ("fork") other processes (*children* processes) dynamically. A parent might share all or some of its resources (address space, open files) with a child process either by choice or mandated by the OS. A (parent) process typically creates children processes to perform tasks for it. There are several reasons why allowing collection of cooperating processes is benefical: information sharing, computational speedup, modularity, and convenience. To be able to cooperate processes must be able to communicate (via some *interprocess-communication* (*IPC*) facility which is done either through shared memory or message passing. Both techniques need some OS support:

1. shared memory needs:
   a. the ability for processes to share memory
   b. synchronization mechanism to coordinate access to shared memory (e.g., semaphores)
2. message passing needs:
   a. send and receive message primitives
   b. buffering of messages or mailboxes, etc.

POSIX Process Management API (used by Linux, UNIX, and Mac OS X)

The **fork** system call is used to create a child process.

process_id = fork( );

The fork makes a copy of the parent process to become the child process. The only difference is that the fork returns value 0 to the child process and the parent receiving the child's process id number (or a negative error number). Both continue execution after the fork statement. Since both processes each have their own address space, changes to variables by either process are local to that process, but both share the parent's open files.

C Fork Example

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

/*   Generating a child process (no error check) */
void main( void ){
  pid_t pid;
  pid = fork( );
  if (pid == 0)
    printf("In the CHILD process\n");
  else
    printf("In the PARENT process\n");
}
```

C++ Fork Example - Figure 7.1 forker.cpp

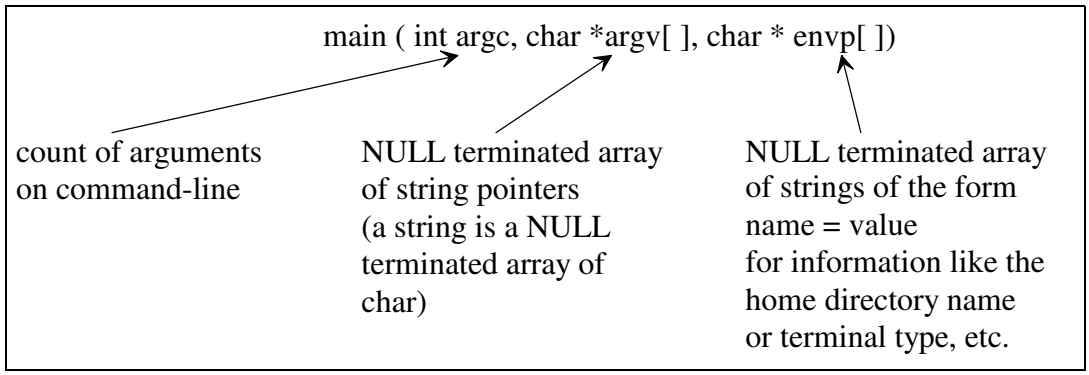
```
#include <unistd.h>
#include <stdio.h>
#include <iostream>
using namespace std;

int main(){
  int loopCount = 5; // each process will get its own loopCount
  cout << "I am still only one process." << endl;
  pid_t returnedValue = fork();
  if(returnedValue < 0){
    // still only one process
    perror("error forking"); // report the error
    return -1;
  } else if (returnedValue == 0){
    // this must be the child process
    while(loopCount > 0){
      cout << "I am the child process." << endl;
      loopCount--; // decrement child's counter only
      sleep(1); // wait a second before repeating
    }
  } else {
    // this must be the parent process
    while(loopCount > 0){
      cout << "I am the parent process; my child's ID is "
           << returnedValue << "." << endl;
      loopCount--; // decrement parent's counter only
      sleep(1);
    }
  }
  return 0;
}
```

========== Output ==========

```
I am still only one process.
I am the parent process; my child's ID is 17066.
I am the child process.
I am the parent process; my child's ID is 17066.
I am the child process.
I am the parent process; my child's ID is 17066.
I am the child process.
I am the parent process; my child's ID is 17066.
I am the child process.
I am the parent process; my child's ID is 17066.
I am the child process.
```

C/C++ programming language aside: command-line arguments

Information can be passed to the main function in the form of command-line arguments and environmental variables:



Typically, an **exec** (of some sort) system call is used after a fork to load/overlay a new binary/executable for the child to run. The **exec** system calls can pass information to the new binary in the form of command-line arguments (either as a list (**l**) or vector (**v**)) and an optional environment variable list (denoted by **e**) envp. A **p** indicates that the current PATH should be used when searching for executable files; otherwise the full path must be specified.

Prototypes of the exec system call

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/);

int execv(const char *path, char *const argv[]);

int execle (const char *path,char *const arg0[],... , const char *argn,
                                                 char * /*NULL*/, char *const envp[]);

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

int execlp (const char *file, const char *arg0, ..., char *argn, char * /*NULL*/);

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

exec Example - Figure 7.3 execer.cpp

```
#include <unistd.h>
#include <stdio.h>
#include <iostream>
using namespace std;

int main(){
  cout << "This is the process with ID " << getpid()
       << ", before the exec." << endl;
  execl("/bin/ps", "ps", "axl", NULL);
  perror("error execing ps");
  return -1;
}
```

========== Partial Output ==========

```
This is the process with ID 17301, before the exec.
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4     0     1     0  20   0  12372    88 -      Ss   ?          29:28 init [2]
5     0     2     0  15  -5      0     0 kthrea S<   ?           0:01 [kthreadd]
1     0     3     2 -100  -      0     0 migrat S<   ?           0:07 [migration/0]
1     0     4     2  15  -5      0     0 ksofti S<   ?           2:27 [ksoftirqd/0]
5     0     5     2 -100  -      0     0 watchd S<   ?           0:37 [watchdog/0]

. . .
4     0 16343  4702  20   0  96040  4716 -      Ss   ?           0:00 sshd: fienup[priv]
5  2194 16354 16343  20   0  96040  3356 -      S    ?           0:00 sshd: fienup@pts/4
0  2194 16355 16354  20   0  60312  4596 rt_sig Ss   pts/4       0:00 -tcsh
0  2194 17301 16355  20   0  10696   836 -      R+   pts/4       0:00 ps axl

. . .
```

Note: exec'ed process runs as same ID as

C programming language aside: parameter passing
In C only pass-by-value exists (no pass-by-reference as in C++)! To have a procedure change the value of a parameter, the parameter's address must be passed (using the & address-of operator). When a procedure is passed a parameter's address, the address can dereference (using the * dereferrencing operator) to manipulate the parameter's value.

The parent process might wait for all or some of its child processes to terminate before it continues execution or might execute concurrently with all of them. In POSIX, a **wait** or **waitpid** system call can be used to wait for a child to finish before the parent continues.

Description of wait and waitpid

```
pid = wait(&status);
```

-1 if no children (errno set to ECHILD) or value of child pid (pid_t)

status of child's exit (int *)

wstat macro can be used

WCOREDUMP(status)
WIFEXITED(status)
WIFSIGNALED(status)
WIFCONTINUED(status)
WIFSTOPPED(status)
WEXITSTATUS(status)
WTERMSIG(status)
WSTOPSIG(status)

Interpretations of status:

1) If process terminated normally,

byte 3	byte 2	byte 1	byte 0
0 not used	0 not used	exit code	0

2) If termination due to uncaught signal,

byte 3	byte 2	byte 1	byte 0
0 not used	0 not used	0	signal number

```
pid = waitpid(pid, &status, options);
```

-1 if no children or value of child pid (pid_t)

-1 any child or specific pid of child to wait for (pid_t)

status of child's exit (int *) wstat macro can be used

0 don't care or bitwise OR of flags to modify wait behavior

fork, exec, and waitid Example - Figure 7.6 microshell.cpp

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <string>
#include <iostream>
using namespace std;

int main(){
  while(1){ // loop until return
    cout << "Command (one word only)> " << flush;
    string command;
    cin >> command;
    if(command == "exit"){
      return 0;
    } else {
      pid_t returnedValue = fork();
      if(returnedValue < 0){
        perror("error forking");
        return -1;
      } else if (returnedValue == 0){
        execlp(command.c_str(), command.c_str(), NULL);
        perror(command.c_str());
        return -1;
      } else {
        if(waitpid(returnedValue, 0, 0) < 0){
          perror("error waiting for child");
          return -1;
        }
      }
    }
  }
}
```

========== Output ==========

```
<52 student:~/cs143/examples/ch07 >./microshell
Command (one word only)> who
algol    pts/1        Mar 28 09:31 (cac114a-01.chas.uni.edu)
fienup   pts/4        Mar 29 06:06 (fienup.cs.uni.edu)
fienup   pts/7        Mar 29 09:38 (fienup.cs.uni.edu)
grahataa pts/9        Mar 28 16:19 (96net1.199.indytel.com)
Command (one word only)> ls
a.out       execer_output.txt  fork.c   forker.cpp         launcher.c
microshell.cpp
execer.cpp  fork-exec.c        fork.c~  forker_output.txt  microshell
multiforker.c
Command (one word only)> date
Thu Mar 29 10:04:58 CDT 2012
Command (one word only)> exit
<53 student:~/cs143/examples/ch07 >
```

(From Lecture 1) Hardware support for Operating Systems

Need protection from user programs that:
1. go into an infinite loop
2. access memory of other programs or the OS
3. access files of other programs

Protection Techniques

1) **CPU Timer** - OS sets a timer to expire and interrupt a user pgm before the user pgm is started. Remember that only one program (in a single CPU system) can be executing at a time so when the OS turns control over to a user program it has "lost control."
   Modifications to the CPU timer are privileged

2) **Dual-Mode Operation** - the CPU has two (or more) modes of operation: user mode and system(/supervisor/monitor/privileged) mode with some privileged instructions only executable in system mode. A mode-bit within the CPU's *processor-status-word* (PSW) register is used to indicate whether the CPU is executing in user or system mode. The set of all machine-language instructions are divided into:
   a) privileged instructions that can only be executed in system mode, and
   b) non-privileged instructions that can be executed in any mode of operation.

   Every time an instruction is executed by the CPU, the hardware checks to see if the instruction is privileged and whether the mode is user. Whenever this case is detected, an exception (internal interrupt/trap) is generated that turns CPU control back over to the operating system.

3) **Restrict a user program to its allocated address space.** In a simple computer, a user program might be allocated a single continguous address space in memory. The two special purpose CPU registers: StartMemory and EndMemory can bracket the user program's address space. All memory addresses that the user program performs can be checked by hardware in the CPU to make sure that they fall between the values in these registers. If the user program tries to access memory outside the range of addresses indicated by these registers, an interrupt/exception is raised to return control back to the operating system. On more complex computers, a memory-management unit (MMU) provides a more sophicated address mapping scheme (paging, segmentation, paged segments, none). Modifications to the memory-management registers are privileged.