CSCE 3600
Principles of Systems Programming

Processes

University of North Texas

Process Management

2

What is a Process?

- A program is a *passive* set of instructions stored on a secondary storage device, such as a disk
- A process is an *active* execution of a program stored in memory
  - Program becomes process when loaded into memory
- Processes can create sub-processes to execute concurrently
- The execution of a process must progress in a sequential fashion
  - The CPU executes one instruction of the process after another until the process completes, or other event occurs

3

More Precise Definition

- A process is the context (i.e., information and data) maintained for an executing program
  - A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task
- Intuitively, a process is the abstraction of a physical processor
  - Exists because it is difficult for the OS to otherwise coordinate many concurrent activities, such as incoming network data, multiple users, etc.

4

Process Management

- The OS is responsible for the following activities
  - Process creation and deletion
  - Process suspension and resumption
  - Provision of mechanisms for:
    - Process synchronization
    - Process communication
  - Deadlock handling
- How does OS correctly run multiple processes concurrently?
  - What kind of information must be kept?
  - What does OS have to do to run processes correctly?
  - OS must be able to distinguish among different processes
    - Multiple programs may be loaded into memory at same time
    - Each process is assigned a unique, non-negative integral process ID, or PID

Process Identifiers

- Special processes with well-known process IDs
  - swapper / sched
    - PID 0, as part of the kernel, a system process responsible for memory management
  - init   replaced by systemd in many Linux distributions
    - PID 1, a continually-running daemon process (i.e., one that runs in the background) responsible for starting up and shutting down the system
    - Invoked by the kernel at the end of the bootstrap procedure

Check out the **`pstree`** command that prints a tree of the processes

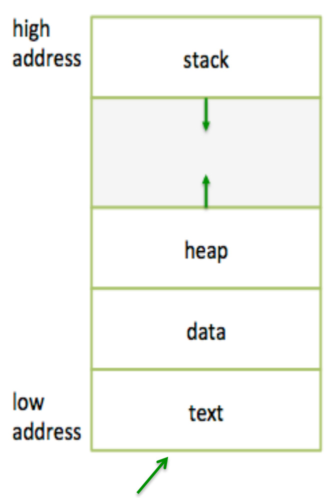
CSE

Process Context

7

Process Context

- When a program loaded into memory, it is organized into the following segments of memory
  - text contains the actual program code, or executable instructions
  - data contains global and static variables initialized at runtime
  - heap contains dynamic memory allocated at runtime
  - stack contains return addresses, function parameters, and variables

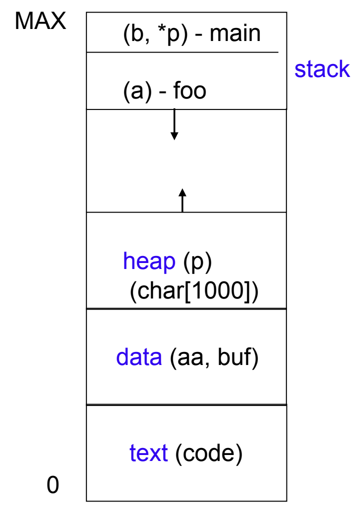


This is known as User Level Context

8

User Level Context

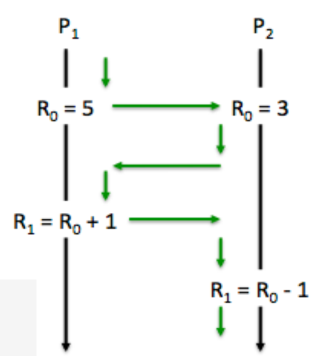
```
…
int aa;
char buf[1000];
void foo() {
  int a;
   …
}
main() {
  int b;
  char *p;
  p = new char[1000];
  foo();
}
```



9

Process Context

- Is the *user level context* sufficient?
  - Does it contain all the states necessary to run a program?
    - Only if the system runs through one program at a time
    - The OS typically needs to switch back and forth between programs – processes must be "swapped" in and out from getting to use the CPU



- R_1 in P_1 is incorrect… why? How to make it right?
  - Save R_0 in P_1 before switching
  - Restore R_0 in P_1 when switching from P2 to P1
- Registers should be a part of process context
  - This is called Register Context

10

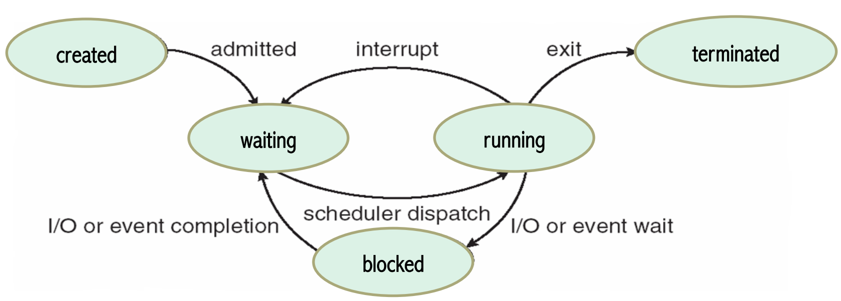
Register Context

- We need to save everything that we need to independently run a process
  - Program Counter (PC)
    - Address of next instruction to be executed (may be in kernel or user memory space of this process)
  - Processor Status Register
    - Contains the hardware status at the time of preemption – contents and format are hardware dependent
  - Stack Pointer (SP)
    - Points to the top of the kernel or user stack, depending on the mode of operation at the time of preemption
  - General-Purpose Registers
    - Hardware dependent, R_0, R_1, R_2, …

11

Process State

- A process executes according to the following state machine:
  - created also "new", initial state when created
  - waiting also "ready", awaiting to be scheduled for execution
  - running actively executing instructions on the CPU
  - blocked unable to continue without event occurring, e.g., I/O
  - terminated no longer running due to completion or being killed



12

Process Context

- User level context
  - Code, data, stack, heap
- Register context
  - R0, R1, …, PC, SP, etc.
- What else is needed?
  - OS resources
    - Open files, signal related data structures, etc.

While a program is executing, the process can be uniquely identified by a number of elements, including:

- Identifier (PID)
- State
- Priority
- Program counter
- Memory pointers
- Context data
- I/O status information
- Accounting information

To run a process correctly, the process instructions must be executed within the process context!

13

Process Context

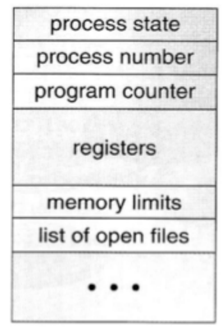
- Where is the *process context* stored?
  - User level context is in memory
  - Other context information is stored in a data structure called process control block
    - Contains other information that the OS needs to manage the process
      - Process status (running, waiting, etc.)
      - Process priority
      - …
  - The OS has a process control block table
    - For each process, there is one entry in the table

14

Process Control Block (PCB)

Information associated with each process (also called task control block)

- Process state
  - Running, waiting, etc.
- Program counter
  - Location of instruction to execute next
- CPU registers
  - Contents of all process-centric registers
- CPU scheduling information
  - Priorities, scheduling queue pointers
- Memory-management information
  - Memory allocated to the process
- Accounting information
  - CPU used, clock time elapsed since start, time limits
- I/O status information
  - I/O devices allocated to process, list of open files



15

CSE

Context Switch

16

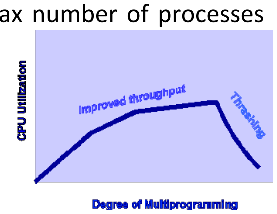
Process Scheduling

- CPU Scheduler (short-term scheduler)
  - Selects which processes should be executed next and allocates the CPU
  - Invoked very frequently (milliseconds)
- Processes can be described as either
  - I/O-bound
    - Spends more time doing I/O than computations; many short CPU bursts
  - CPU-bound
    - Spends more time doing computations; few very long CPU bursts

17

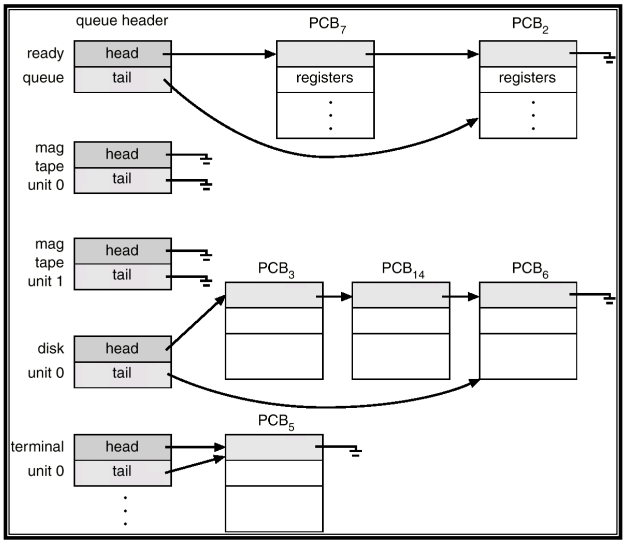
Process Scheduling

- Job Scheduler (long-term scheduler)
  - Selects which processes to be brought into ready queue
  - Invoked very infrequently (seconds, minutes)
  - Controls degree of multiprogramming, the max number of processes accommodate efficiently
- Maintains scheduling queues of processes
  - Job queue
    - Set of all PCBs in the system
  - Ready queue
    - Set of all processes residing in main memory, ready and waiting to execute
  - Device queues
    - Set of processes waiting for an I/O device
  - Processes migrate among the various queues



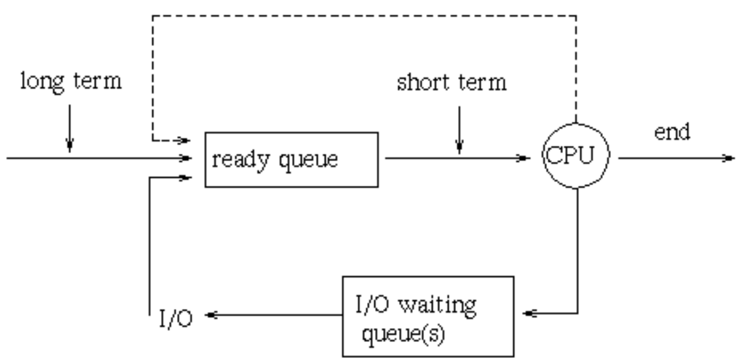
18

Ready Queue and I/O Device Queues



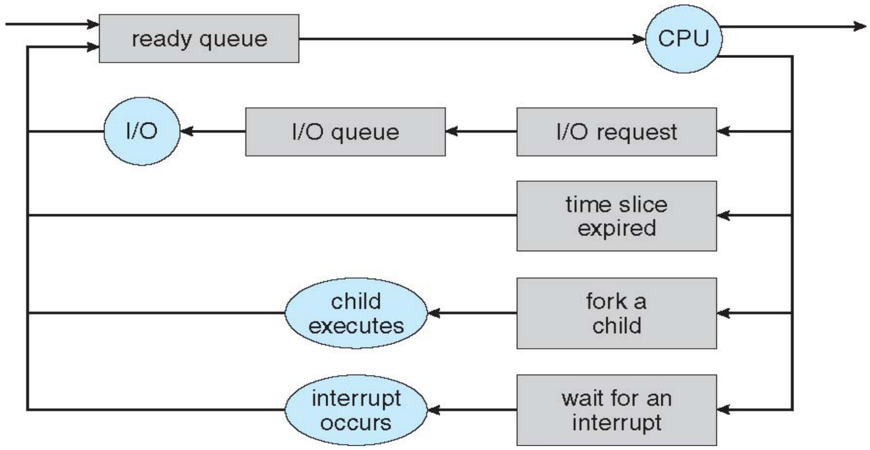
19

Long- and Short-Term Schedulers



20

Process Scheduling Representation



Queueing diagram represents queues, resource flows

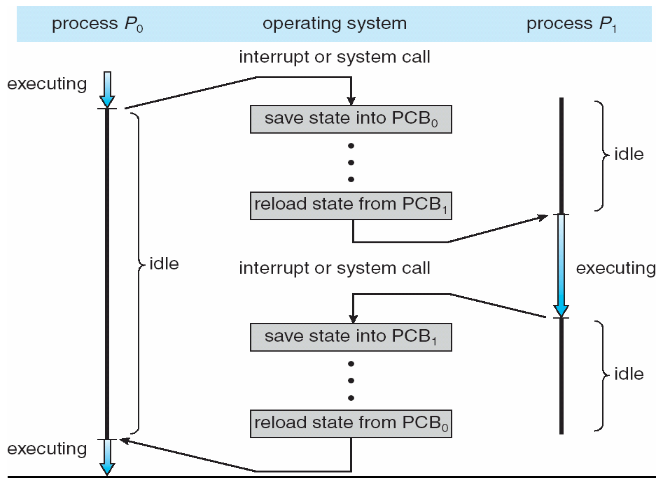
21

Context Switch

- When CPU switches to another process, the OS must
  - Save the PCB of old process being swapped out
  - Select new process to be swapped in
  - Load the PCB of the new process being swapped in
- Context-switch time is overhead
  - The system does no useful work while switching
    - Typical time about 1 μsec
  - The more complex the OS and the PCB, the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU, resulting in multiple contexts loaded at once

22

Context Switch



23

CSE

Process Execution

24

The `exec()` System Call

- Calling one of the `exec()` family of system calls
  - Terminates the currently running program, and
  - Replaces it with a new specified program that starts executing in its `main()` function
- The process ID does not change across an `exec` because a new process is *not* created
- `exec()` merely replaces the current process (its text, data, heap, and stack segments) with a brand new program from disk

The `exec()` Family

- There are 6 versions of the `exec` function, and they all do about the same thing:
  - Main difference is how parameters are passed

```
#include <unistd.h>
int execlp(char *file, char *arg0,
      char *arg1, ..., (char *)0);

execlp("sort", "sort", "-n",
            "city", (char *)0);
```

Same as "sort -n city"

The exec() Family

`int execl(const char *path, const char *arg, ... );`

- `execl` takes full path name of command and variable length of arguments terminated by `NULL`

```
execl("/bin/ls", "/bin/ls", "-r", "-t", "-l", NULL);
```

`int execlp(const char *file, const char *arg, ... );`

- `execlp` will try to find the command from `$PATH`, so full path to command not needed

```
execlp("ls", "ls", "-r", "-t", "-l", NULL);
```

`int execle(const char *path, const char *arg, ..., char *const envp[] );`

- `execle` uses an argument list and environment variables

```
char *env[] = {"PATH=/bin", NULL};
execle("child.exe", "child", "arg1", "arg2", NULL, env);
```

27

The exec() Family

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

- `execv` is the equivalent of `execl`, except that the arguments are passed in as a NULL terminated array

```
char *args[] = {"/bin/ls", "-r", "-t", "-l", NULL};
execl("/bin/ls", args);
```

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

- `execvp` is the equivalent of `execlp`, except that the arguments are passed in as a NULL terminated array

```
char *args[] = {"ls", "-r", "-t", "-l", NULL};
execvp("ls", args);
```

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

28

The exec() Family

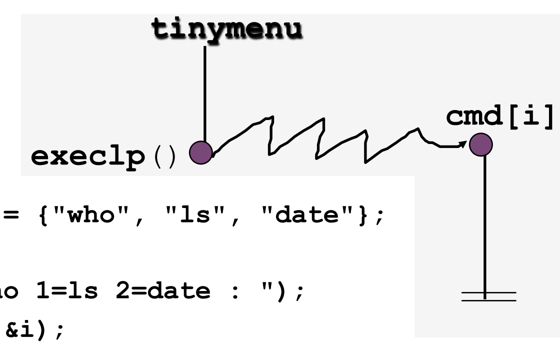
- All six return –1 on error, but no return on success
- Accept either a pathname or filename argument
- Command-line arguments are specified as separate arguments or we have to build an array of pointers to the arguments and pass the address of the array

29

execlp Example

```
#include <stdio.h>
#include <unistd.h>

int main()
{
        char *cmd[] = {"who", "ls", "date"};
        int i;
        printf("0=who 1=ls 2=date : ");
        scanf("%d", &i);
        execlp(cmd[i], cmd[i], (char *)0 );
        printf("execlp failed\n");
        return 0;
}
```



printf() not executed unless there is a problem with execlp()

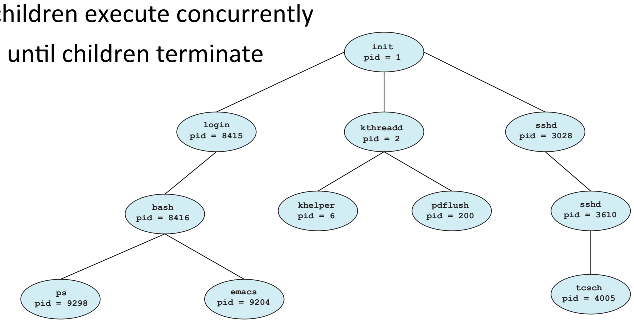
30

Process Creation

31

Process Creation

- A parent process creates child processes, which create other processes, forming a tree of processes
  - Generally, processes identified and managed via a process identifier (PID)
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate



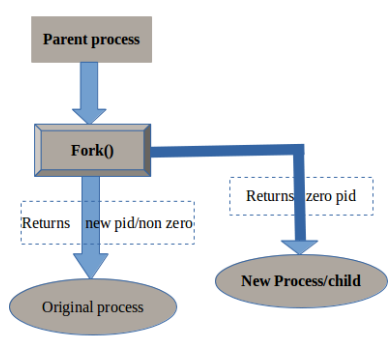
32

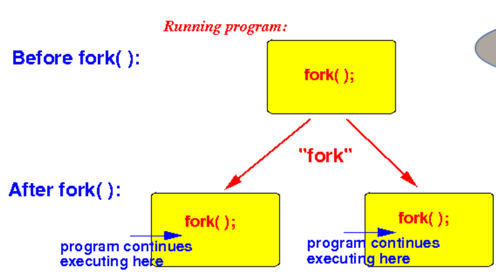
The `fork()` System Call

- Creation of a *new* process accomplished using the `fork()` system call
  - `fork()` creates a child process by making an *exact copy* of the parent process *and* then starts the process concurrently
  - `fork()` system call is unique
    - Called once, but returns twice with the parent receiving the child's unique PID and the child receiving 0 as the return value from `fork()`
  - The process that initiates the `fork()` becomes the parent process of the newly created child process
    - The child process inherits a copy of the parent's memory space, but they do *not* share the same memory as both parent and child processes will execute in their own environments

The `fork()` System Call

- Current process splits into two processes: parent, child
  - Returns -1 if unsuccessful
  - Returns 0 in the child
  - Returns the child's PID in the parent





- Execution continues in both processes

The `fork()` System Call

- There are two uses for fork:
  - When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time
    - This is common for network servers – the parent waits for a service request from a client
    - When the request arrives, the parent calls `fork()` and lets the child handle the request
    - Parent goes back to waiting for the next service request to arrive
  - When a process wants to execute a different program
    - This is common for shells
    - In this case, the child typically does an `exec()` right after it returns from the fork

35

The `fork()` System Call

- The two main reasons for fork to fail are
  - If there are already too many processes in the system (which usually means something else is wrong)
  - If the total number of processes for this real user ID exceeds the system's limit

36

fork() Example

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

int main()
{
        pid_t pid;          /* could be int */
        int i;
        pid = fork();
        printf("PID=%d\n", pid);
        if( pid > 0 )
        {
                /* parent */
                for( i=0; i < 10; i++ )
                        printf("\t\t\tPARENT %d\n", i);
        }
        else
        {
                /* child */
                for( i=0; i < 10; i++ )
                        printf("CHILD %d\n", i );
        }

        return 0;
}
```

37

fork() Example

- Notes for code on previous slide
  - `i` is copied between parent and child
  - The switching between the parent and child depends on many factors:
    - Machine load, system process scheduling
  - I/O buffering affects amount of output shown
  - Output interleaving is *nondeterministic*
    - Cannot determine exact output by looking at code

38

Process Synchronization

39

wait() and waitpid() System Calls

- The `wait()` and `waitpid()` system calls force the parent process to suspend execution until the child process has completed
  - `waitpid()` waits for a specific child process identified by its PID while `wait()` simply waits for the first child process to terminate (if the parent has more than one child process)
  - Both return the PID of the terminated process if successful
    - Or –1 if an error occurred (usually means no child exists to wait on)
  - Once the child process has terminated, the parent process resumes execution

40

`wait()` Function

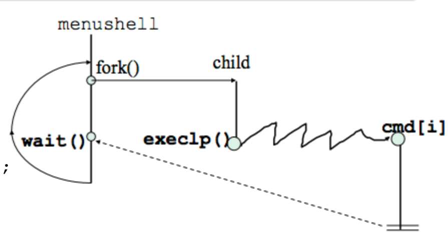
```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statloc);
```

- **`statloc`** can be `(int *)0` or a variable which will be bound to status information about the child
- A process that calls `wait()` can
  - suspend (block) if all of its children are still running
  - return immediately with the termination status of a child
  - return immediately with an error if there are no child processes.

`wait()` Example

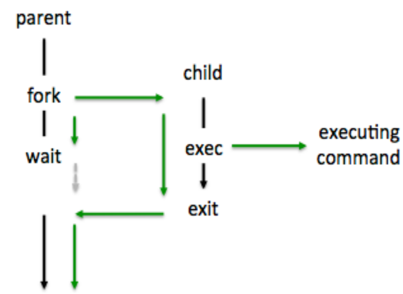
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    char *cmd[] = {"who", "ls", "date"};
    int i;
    while( 1 ) {
        printf("0=who 1=ls 2=date : ");
        scanf("%d", &i);
        if(fork() == 0) {
            /* child */
            execlp( cmd[i], cmd[i], (char *)0 );
            printf("execlp failed\n");
            exit(1);
        }
        else {
            /* parent */
            wait( (int *)0 );
            printf("child finished\n");
        }
    } /* while */
} /* main */
```



The `fork-exec` Model

- Common use of the `fork()` and `exec()` system calls centers around being able to run another program in parallel without *terminating* the current process
  - Parent process uses `fork()` to create the child process, which will then use the `exec()` family of system calls to run the desired program while the parent waits on the child to terminate



CSE

Process Termination

Process Termination

- A process may be terminated
  - When it executes its last statement and asks the operating system to delete it
  - By calling the `exit()` system call in the `<stdlib.h>` library
- A parent may terminate execution of child processes using the `abort()` system call if
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - Parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
- When a child process terminates, a SIGCHLD signal is sent to the parent

45

Zombies and Orphans

- A child process whose parent has terminated is referred to as an orphan
  - Child is still executing, but parent has terminated
  - Some process is needed to query the child's exit status
- When a child exits and its parent is not currently waiting (i.e., executing a `wait()`), it becomes a zombie
  - A zombie is not really a process (since it terminated), but the system still has an entry in the process table for the non-existing child process
- When a parent terminates, any orphans and zombies are adopted by the init process (PID 1) of the system

46

CSE

Process Identification

Process Identification

- `pid = getpid( );`
  - Returns its own process id
- `pid = getppid( );`
  - Returns parent process id
- `uid = getuid( );`
  - Returns real user ID of own process
- `newpg = setpgrp( );`
  - Sets process group of own process to itself
- `pgid = getpgrp( );`
  - Returns the process group ID of own process

Process and Group ID’s Example

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
//#include <unistd.h>
int main() {
     pid_t cpid, pid, pgid, cpgid; //process id's and process groups
     cpid = fork();
     if (cpid == 0) {
          /* CHILD */
          //set process group to itself
          setpgrp(); //<---------------------------!
          //print the pid, and pgid of child from child
          pid = getpid();
          pgid = getpgrp();
          printf("Child:          pid:%d pgid:*%d*\n", pid, pgid);
     }
     else if (cpid > 0) {
          /* PARENT */
          //set the process group of child
          setpgid(cpid, cpid); //<----needed to disambiguate runtime process
          //print the pid, and pgid of parent
          pid = getpid();
          pgid = getpgrp();
          printf("Parent:         pid:%d pgid: %d \n", pid, pgid);
          //print the pid, and pgid of child from parent
          cpgid = getpgid(cpid);
          printf("Parent: Child's pid:%d pgid:*%d*\n", cpid, cpgid);
     }
     else {
          /*ERROR*/
          perror("fork");
          exit(1);
     }
     return 0;
}
```

With this, it will not matter which runs first, parent or child, as the result will be the same – the child placed in the appropriate process group

49

Process Data

50

Process Data

- Since a child process is a copy of the parent, it has *copies* of the parent's data
- A change to a variable in the child will *not* change that variable in the parent

Process Data Example

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int glbvar = 6;
int main() {
    int locvar = 88;
    pid_t pid;
    printf("Before fork()\n");
    if( (pid = fork()) == 0 ) {
        /* child */
        glbvar++;
        locvar++;
    }
    else if ( pid > 0 ) {
        /* parent */
        sleep(2);
    }
    else
        perror("fork error");
    printf("pid=%d, glbvar=%d, locvar=%d\n", getpid(), glbvar, locvar);
    return 0;
} /* end main */
```

Inherited Data and File Descriptors

- A forked child has instances of current values of the variables and open file descriptors
- Variables
  - Passed by value (i.e., a copy)
- Read/write pointers for a file
  - Passed by reference

53

Process File Descriptors

- A child and parent have copies of the file descriptors, but the R-W pointer is maintained by the system
  - The R-W pointer is shared
- This means that a `read()` or `write()` in one process will affect the other process since the R-W pointer is changed

54

Process File Descriptors Example

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
void printpos(char *msg, int fd);
int main() {
    int fd; /* file descriptor */
    pid_t pid;
    char buf[10]; /* for file data */
    if ((fd=open("file1", O_RDONLY)) < 0)
        perror("open");
    read(fd, buf, 10); /* move R-W ptr */
    printpos("Before fork", fd );
    if( (pid = fork()) == 0 ) {
        /* child */
        printpos("Child before read", fd);
        read(fd, buf, 10);
        printpos("Child after read", fd);
    }
    else if( pid > 0 ) {
        /* parent */
        wait((int *)0);
        printpos("Parent after wait", fd);
    }
    else
        perror("fork");
}
```

55

Process File Descriptors Example

```
/* Print position in file */
void printpos(char *msg, int fd) {
    long int pos;
    if( (pos = lseek( fd, 0L, SEEK_CUR) ) < 0L )
        perror("lseek");
    printf("%s: %ld\n", msg, pos);
}
```

```
$ ./a.out
Before fork: 10
Child before read: 10
Child after read: 20
Parent after wait: 20
```

what's happened?

56