UNIX System Programming

Processes

- ❖ Objectives
  - – look at how to program UNIX processes
  - – fork( ), exec( ), wait( )

Overview

1. What is a Process?
2. `fork()`
3. `exec()`
4. `wait()`
5. Process Data
6. File Descriptors across Processes
7. Special Exit Cases
8. IO Redirection
9. getenv/putenv, ulimit()

1. What is a Process?

- A process is an executing program.

- A process:

```
$ cat file1 file2 &
```

- Two processes:

```
$ ls | wc - l
```

- Each user can run many processes at once (e.g. using &)

A More Precise Definition

- A process is the *context* (the information/data) maintained for an executing program.
- 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.
- IMPORTANT: A process is **sequential**

What makes up a Process?

- program code
- Values in machine registers
- global data
- stack
- open files (file descriptors)
- an environment (environment variables; credentials for security)

Some of the Context Information

– Process ID (`pid`)	unique integer
– Parent process ID (`ppid`)	
– Real User ID	ID of user/process which started this process
– Effective User ID	ID of user who wrote the process' program
– Current directory	
– File descriptor table	
– Environment	`VAR=VALUE` pairs

continued

– Pointer to program code	
– Pointer to data	Memory for global vars
– Pointer to stack	Memory for local vars
– Pointer to heap	Dynamically allocated

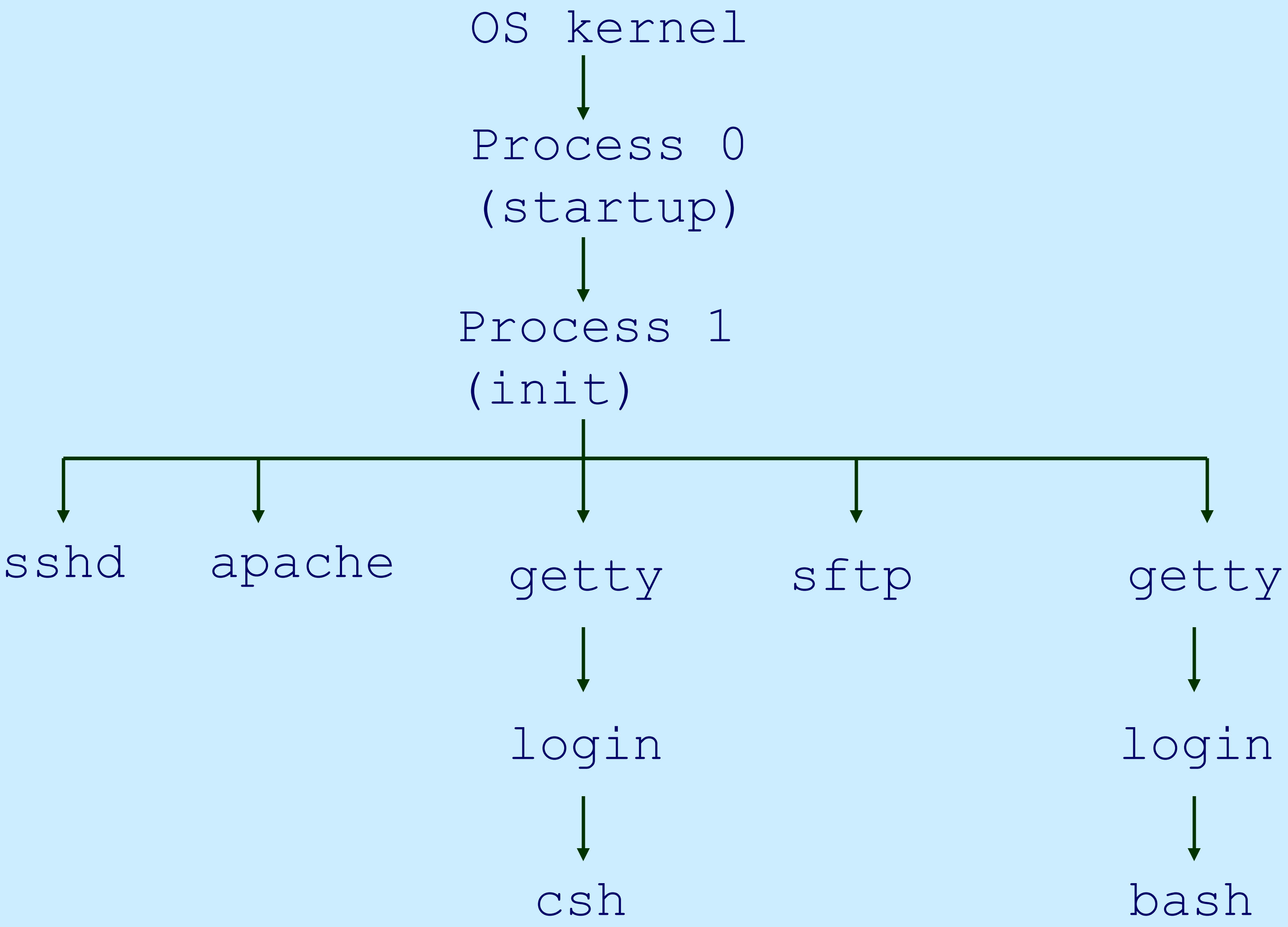
– Execution priority

– Signal information

Important System Processes

- init/systemd – Mother of all processes. init is started at boot time and is responsible for starting other processes.
  - init uses file inittab & directories: /etc/rc?.d
- getty – login process that manages login sessions.

Unix Start Up Processes Diagram



OS kernel

Process 0 (startup)

Process 1 (init)

sshd

apache

getty

login

csh

sftp

getty

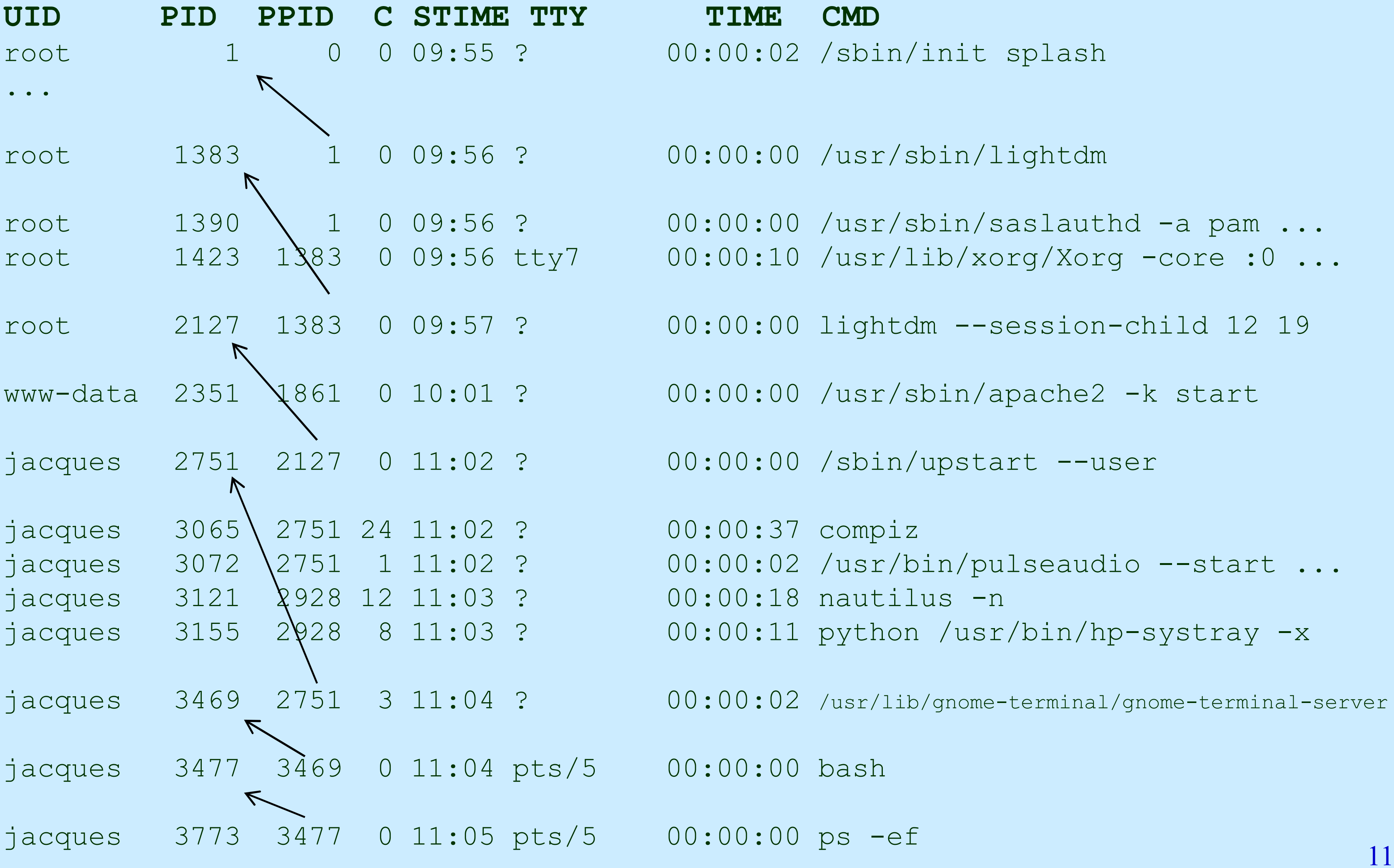
login

bash

ps –ef

```
UID       PID  PPID C STIME TTY      TIME   CMD
root        1     0  0 09:55 ?       00:00:02 /sbin/init splash
root        2     0  0 09:55 ?       00:00:00 [kthreadd]
root        3     2  0 09:55 ?       00:00:00 [ksoftirqd/0]
root        5     2  0 09:55 ?       00:00:00 [kworker/0:0H]
root       10     2  0 09:55 ?       00:00:00 [watchdog/0]
root       17     2  0 09:55 ?       00:00:00 [netns]
root       18     2  0 09:55 ?       00:00:00 [perf]
root       23     2  0 09:55 ?       00:00:00 [crypto]
root       79     2  0 09:55 ?       00:00:00 [scsi_eh_0]
root      103     2  0 09:55 ?       00:00:00 [charger_manager]
root      275     2  0 09:55 ?       00:00:00 [kworker/1:1H]
root     1089     1  0 09:56 ?       00:00:00 /lib/systemd/systemd-logind

root     1121     1  0 09:56 ?       00:00:00 /usr/lib/bluetooth/bluetoothd
root     1199     1  0 09:56 ?       00:00:00 /usr/sbin/NetworkManager --no-daemon
root     1210     1  0 09:56 ?       00:00:00 /usr/sbin/cron -f
root     1214     1  0 09:56 ?       00:00:00 /usr/sbin/thermald --no-daemon --dbus-enable
syslog   1218     1  0 09:56 ?       00:00:00 /usr/sbin/rsyslogd -n
root     1238     1  0 09:56 ?       00:00:00 /usr/lib/policykit-1/polkitd --no-debug
colord   1284     1  0 09:56 ?       00:00:00 /usr/lib/colord/colord
root     1320     1  0 09:56 ?       00:00:00 /usr/sbin/irqbalance --pid=/var/run/irqbalance.pid

root     1383     1  0 09:56 ?       00:00:00 /usr/sbin/lightdm

root     1390     1  0 09:56 ?       00:00:00 /usr/sbin/saslauthd -a pam -c -m /var/spool/postfix...
root     1423  1383  0 09:56 tty7    00:00:10 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/...
root     1441     1  0 09:56 ?       00:00:00 /usr/sbin/sshd -D
mysql    1450     1  0 09:56 ?       00:00:04 /usr/sbin/mysqld
root     1475  1199  0 09:56 ?       00:00:00 /sbin/dhclient -d -q -sf /usr/lib/NetworkMan...
root     1772     1  0 09:56 tty1    00:00:00 /sbin/agetty --noclear tty1 linux
root     1861     1  0 09:56 ?       00:00:00 /usr/sbin/apache2 -k start

root     2127  1383  0 09:57 ?       00:00:00 lightdm --session-child 12 19
```

What happens when I issue a command? Ancestry in UNIX….



```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 09:55 ?        00:00:02 /sbin/init splash
...

root      1383     1  0 09:56 ?        00:00:00 /usr/sbin/lightdm

root      1390     1  0 09:56 ?        00:00:00 /usr/sbin/saslauthd -a pam ...
root      1423  1383  0 09:56 tty7     00:00:10 /usr/lib/xorg/Xorg -core :0 ...

root      2127  1383  0 09:57 ?        00:00:00 lightdm --session-child 12 19

www-data  2351  1861  0 10:01 ?        00:00:00 /usr/sbin/apache2 -k start

jacques   2751  2127  0 11:02 ?        00:00:00 /sbin/upstart --user

jacques   3065  2751 24 11:02 ?        00:00:37 compiz
jacques   3072  2751  1 11:02 ?        00:00:02 /usr/bin/pulseaudio --start ...
jacques   3121  2928 12 11:03 ?        00:00:18 nautilus -n
jacques   3155  2928  8 11:03 ?        00:00:11 python /usr/bin/hp-systray -x

jacques   3469  2751  3 11:04 ?        00:00:02 /usr/lib/gnome-terminal/gnome-terminal-server

jacques   3477  3469  0 11:04 pts/5    00:00:00 bash

jacques   3773  3477  0 11:05 pts/5    00:00:00 ps -ef
```

Finding your own ancestry

You can find the ancestry of your current process by typing:

```
ps -aef --forest | grep -e jacques
```

replace jacques with your username (or the first 7 letters if your name is longer). You should wind up with something similar to this:

```
[jacques@loki 3380]$ ps -aef --forest | grep -e jacques
root       504   919  0 Feb11 ?        00:00:00  \_ sshd: jacques [priv]
jacques    532   504  0 Feb11 ?        00:00:07  |   \_ sshd: jacques@pts/4
jacques    533   532  0 Feb11 pts/4    00:00:00  |       \_ -bash
jacques   4135   533  0 09:47 pts/4    00:00:00  |           \_ ps -aef --forest
jacques   4136   533  0 09:47 pts/4    00:00:00  |           \_ grep --color=auto -e jacques
```

The 2nd column is the process ID (PID)
The 3rd column is the parent process ID (PPID).

Walk the parent tree up to the ssh session.

Tracing processes other than the current one

I wrote my own program to trace the ancestry of a process on Loki:

```
[jacques@loki 3380]$ ./process_tree.pl process_tree
Looking for the ancestry of the process: process_tree

root        919     1  0 Feb07 ?        00:00:55 /usr/sbin/sshd -D
root        504   919  0 Feb11 ?        00:00:00  \_ sshd: jacques [priv]
jacques     532   504  0 Feb11 ?        00:00:07  |   \_ sshd: jacques@pts/4
jacques     533   532  0 Feb11 pts/4    00:00:00  |       \_ -bash
jacques    4443   533  0 09:51 pts/4    00:00:00  |           \_ /usr/bin/perl ./process_tree.pl process_tree
[jacques@loki 3380]$
```

We’ll use this in a few minutes…

Pid and Parentage

- A process ID or *pid* is a positive integer that uniquely identifies a running process, and is stored in a variable of type *pid_t*.

- You can get the process pid or parent’s pid

```
#include <sys/types>
main()
  {
  pid_t pid, ppid;
  printf( "My  PID is:%d\n\n",(pid = getpid()) );
  printf( "Par PID is:%d\n\n",(ppid = getppid()) );
  }
```

Checking out Process IDs

```
jacquesabeland@frigg:~/3380> ./pids
My  PID is:6935

Par PID is:6782

jacquesabeland@frigg:~/3380>  ps -ef | grep -e "6782"
1149      6782  6781  0 08:13 pts/0    00:00:00 -bash
1149      6943  6782  0 08:19 pts/0    00:00:00 ps -ef
1149      6944  6782  0 08:19 pts/0    00:00:00 grep -e 6782

jacquesabeland@frigg:~/3380>  ps -ef | grep -e "6935"
1149      6951  6782  0 08:19 pts/0    00:00:00 grep -e 6935
jacquesabeland@frigg:~/3380>
```

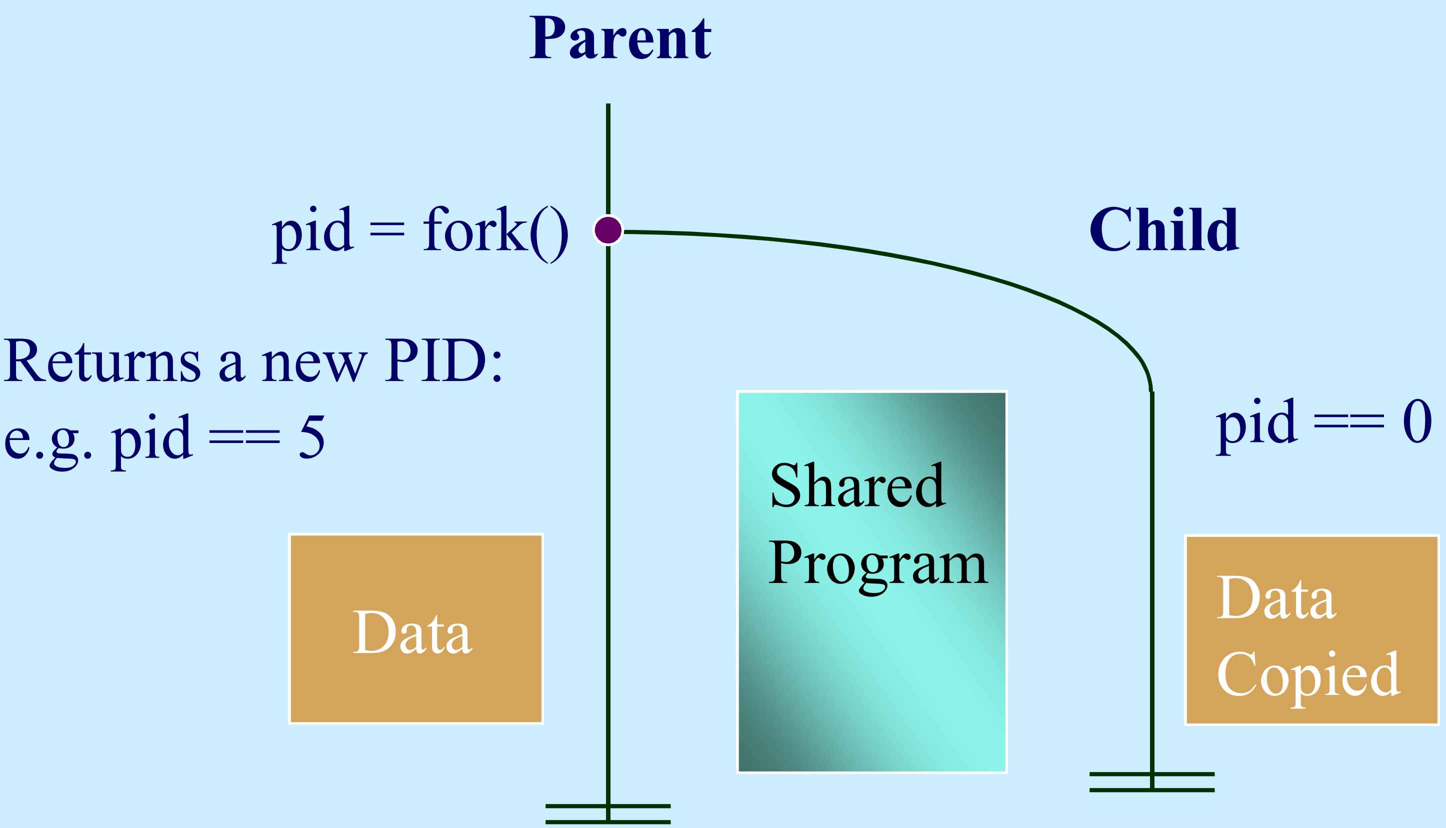
2. fork()

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork( void );
```

- Creates a child process by making a **copy** of the parent process --- an **exact** duplicate or CLONE.
  - Implicitly specifies code, registers, stack, data, file descriptors…
  - The ONLY difference is that the newly created process MUST have a unique process ID for the O/S to schedule it

- Notice the function fork returns a value of type pid_t
- Both the child *and* the parent continue running.

fork() as a diagram



Parent

pid = fork()

Child

Returns a new PID:
e.g. pid == 5

pid == 0

Shared Program

Data

Data Copied

Process IDs (pids revisited)

- `pid = fork();`

- In the child: `pid == 0;`
  In the parent: `pid` == the process ID of the child.

- A program almost always uses this `pid` difference to do different things in the parent and child.

fork() Example

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

int main()
     {
     pid_t pid;
     printf(“Calling fork()\n”);
     pid = fork();
     if( pid == 0 )
             printf(“I’m the child\n”);
     else if( pid > 0)
             printf(“I’m the parent, child has pid %d\n”, pid);
     else
             printf( “Fork returned error code … no child\n”);
     return 0;
     }
```

```
Calling fork()
I'm the child
I'm the parent, child has pid 768
```

fork() Example - Personal preference

```
int main()
 {
   pid_t pid;
   pid_t finishedPID;
   int status, exit_status;

   pid = fork();   // create a process fork here, if we can

   switch(pid)
    {
     case -1: printf("Error forking the process for the first child\n");
              exit(1);
              break;
     case 0:  printf("\tFork was successful. First child code now starting\n");
              exit_status = ChildProcess();
              printf("\tFirst Child has returned from the subroutine. Status=%d\n",exit_status);
              break;
     default:
              printf("PARENT: fork() worked. Child PID=%d\n",pid);
              finishedPID = wait( (int *)0 );  // wait for any child to finish
              printf("PARENT: After the wait, process id %d finished.\n\n\n",finishedPID);
     } // end of switch on first fork()
  exit(0);

} /* main */
```

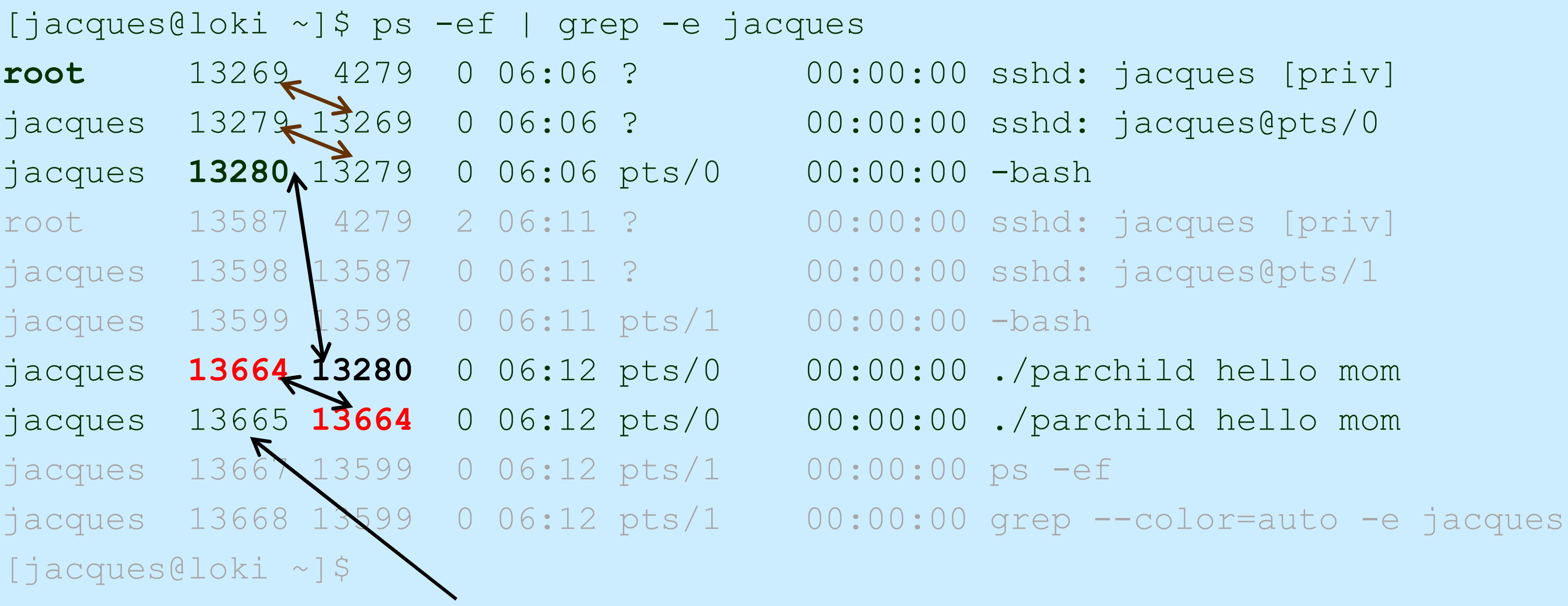
fork() Example   (parchld.c)

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

int main()
{
   pid_t pid, mpid, ppid;         /* could be int */
   int i;

   printf( "My  PID is:%d\n\n",(mpid = getpid()) );
   printf( "Par PID is:%d\n\n",(ppid = getppid()) );

   pid = fork();
   printf("After the fork, the returned PID=%d\n",pid);

   switch(pid)
    {
       case -1: printf("Error forking the process for the first child\n");
                exit(1);
                break;

        case 0: for( i=0; i < 1000; i++ ) {printf( “CHILD %d\n”, i );}
                break;
       default: for( i=0; i < 1000; i++ ) {printf(“\t\t\tPARENT %d\n”, i);  }

     } // end of switch on fork()

    return 0;

  } // end of main()
```

Output: Slightly modified code…

```
[jacques@loki fork_processes]$ gcc -o parchild parchild.c
[jacques@loki fork_processes]$ ./parchild hello mom
My  PID is:13664

Par PID is:13280

After the fork, the returned PID=13665
                        PARENT 0
After the fork, the returned PID=0
CHILD 0
                        PARENT 1
CHILD 1
CHILD 2
                        PARENT 2
CHILD 3
                        PARENT 3
CHILD 4
                        PARENT 4
CHILD 5
                        PARENT 5
CHILD 6
                        PARENT 6
```

The Processes….



```
[jacques@loki ~]$ ps -ef | grep -e jacques
root      13269  4279  0 06:06 ?        00:00:00 sshd: jacques [priv]
jacques   13279 13269  0 06:06 ?        00:00:00 sshd: jacques@pts/0
jacques   13280 13279  0 06:06 pts/0    00:00:00 -bash
root      13587  4279  2 06:11 ?        00:00:00 sshd: jacques [priv]
jacques   13598 13587  0 06:11 ?        00:00:00 sshd: jacques@pts/1
jacques   13599 13598  0 06:11 pts/1    00:00:00 -bash
jacques   13664 13280  0 06:12 pts/0    00:00:00 ./parchild hello mom
jacques   13665 13664  0 06:12 pts/0    00:00:00 ./parchild hello mom
jacques   13667 13599  0 06:12 pts/1    00:00:00 ps -ef
jacques   13668 13599  0 06:12 pts/1    00:00:00 grep --color=auto -e jacques
[jacques@loki ~]$
```

Child PID

Another fork example

in /home/COIS/3380/sample_code/fork_processes
you will find: fork_example.c

When you run it, you'll get somehting like:

```
[jacques@loki 3380]$ ./fork

In the main program, just before the fork() statement
        my process ID is 7004 and my parent's process ID is 27721

PARENT: fork() worked. Child PID=7005
        Fork was successful. First child code now starting
        Child is running… my PID=7005
        ... and my Parent's PID is 7004
        Child is waking up from its sleep...
        First Child has returned from the subroutine. Status=0
PARENT: After the long wait, process id 7005 finished processing.
```

The view from the O/S side

```
[jacques@loki 3380]$ ./process_tree.pl fork
Looking for the ancestry of the process: fork

root       919     1  0 Feb07 ?        00:00:55 /usr/sbin/sshd -D
root     27704   919  0 Feb11 ?        00:00:00  \_ sshd: jacques [priv]
jacques  27720 27704  0 Feb11 ?        00:00:03  |   \_ sshd: jacques@pts/0
jacques  27721 27720  0 Feb11 pts/0    00:00:00  |       \_ -bash
jacques   7123 27721  0 10:27 pts/0    00:00:00  |           \_ ./fork
jacques   7124  7123  0 10:27 pts/0    00:00:00  |               \_ ./fork
[jacques@loki 3380]$
```

Notice the 2nd instance of the fork program is grafted off of the first instance. It is therefore the child process!

Things to Note

- `i` is **copied** between parent and child.
- The CPU switching between the parent and child processes depends on many factors:
  - machine load, system process scheduling
- I/O buffering effects amount of output shown.
- Output interleaving is *nondeterministic*
  - cannot determine output by looking at code

Things to Note

- So far, all we’ve managed to do is make a complete duplicate of all of our code.
- Not very useful.
- *fork* is useful in connection with other systems calls such as *exec*
- *exec* allows you to replace your current running image, with any other application you have access to!

Things to Note

Why?

- It is often desirable to have independent blocks of code written for specific tasks.
- This is the UNIX philosophy of writing things once and reusing that program over and over again.
- Your code may want to be used as a controller which dispatches specific tasks which are contained in, or satisfied by, other programs

Managing resources like HTTP Servers

```
[jacques@loki fork_processes]$ ps -ef | grep -e http
UID        PID  PPID  C STIME TTY          TIME CMD
jacques  14549 13280  0 06:35 pts/0    00:00:00 grep --color=auto -e http
root     28536     1  0 Sep18 ?        00:04:11 /usr/sbin/httpd -DFOREGROUND
apache   37086 28536  0 Oct16 ?        00:00:17 /usr/sbin/httpd -DFOREGROUND
apache   37193 28536  0 Oct16 ?        00:00:11 /usr/sbin/httpd -DFOREGROUND
apache   37249 28536  0 Oct16 ?        00:00:09 /usr/sbin/httpd -DFOREGROUND
apache   37325 28536  0 Oct16 ?        00:00:10 /usr/sbin/httpd -DFOREGROUND
apache   37326 28536  0 Oct16 ?        00:00:05 /usr/sbin/httpd -DFOREGROUND
apache   37328 28536  0 Oct16 ?        00:00:09 /usr/sbin/httpd -DFOREGROUND
apache   38135 28536  0 Oct16 ?        00:00:08 /usr/sbin/httpd -DFOREGROUND
apache   38490 28536  0 Oct16 ?        00:00:06 /usr/sbin/httpd -DFOREGROUND
apache   41674 28536  0 Oct16 ?        00:00:02 /usr/sbin/httpd -DFOREGROUND
apache   41691 28536  0 Oct16 ?        00:00:03 /usr/sbin/httpd -DFOREGROUND
```

exec()

- The exec() calls forces the O/S to load a different images into the current process space.
- This implies that all instructions and data pertaining to the current running image are removed from the process.
- A completely new image, data, stack, program pointer are then created and loaded into memory
- On of the only thing that remains is the process ID created by the O/S fort he original process. This allows the parent process to monitor the child's lifecycle.

3. exec()

- Family of functions for replacing process’s program with the one inside the `exec()` call.

  e.g.

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

execlp(“sort”, “sort”, “-n”,
                  “foobar”, (char *)0);
```

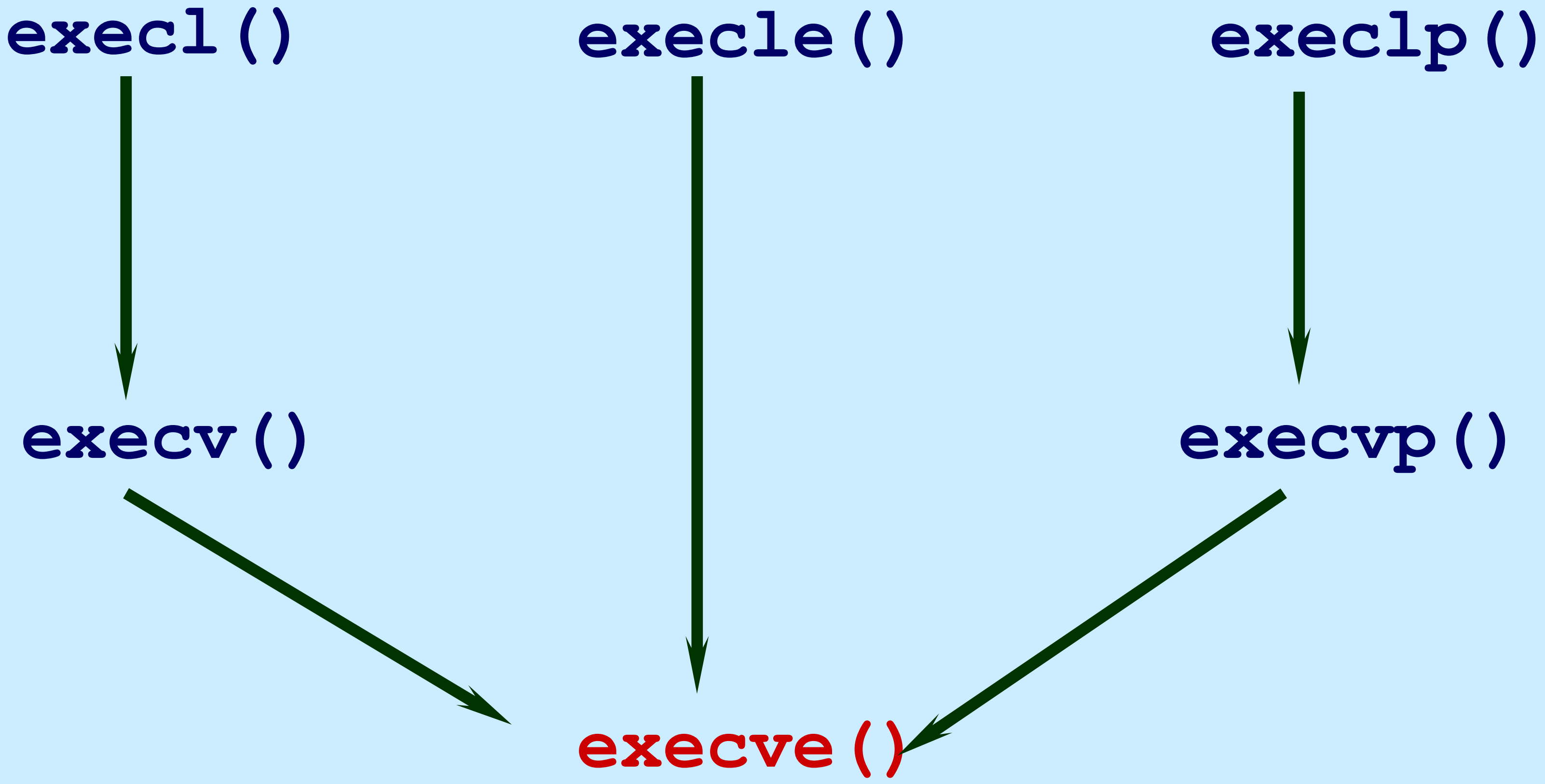
Same as "sort -n foobar"

exec(..) Family

- There are 6 versions of the exec function, and they all do about the same thing: they replace the current program with the text of the new program. Main difference is how parameters are passed.

```
int execl( const char *path, const char *arg0, ...,
           const char *argn, (char *)0 );
int execlp( const char *file, const char *arg0, ...,
           const char *argn, (char *)0 );
int execle( const char *path, const char *arg0, ...,
           const char *argn, char *const envp[] );
int execv( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
int execve( const char *filename, char *const argv [],
            char *const envp[] );
```

exec(..) Family



execl()

execle()

execlp()

execv()

execvp()

execve()

exec_ _ naming convention

- if there's a "p" in the function name, the PATH variable is used to find the first occurrence of the program name. The system will use that binary.

- if there's an "*ℓ*" in the function name, the exec() function expects a list of string values which represent argv[0], argv[1]… as if they were on a command line. The list MUST be terminated by a NULL string (char *)NULL.

exec_ _ naming convention

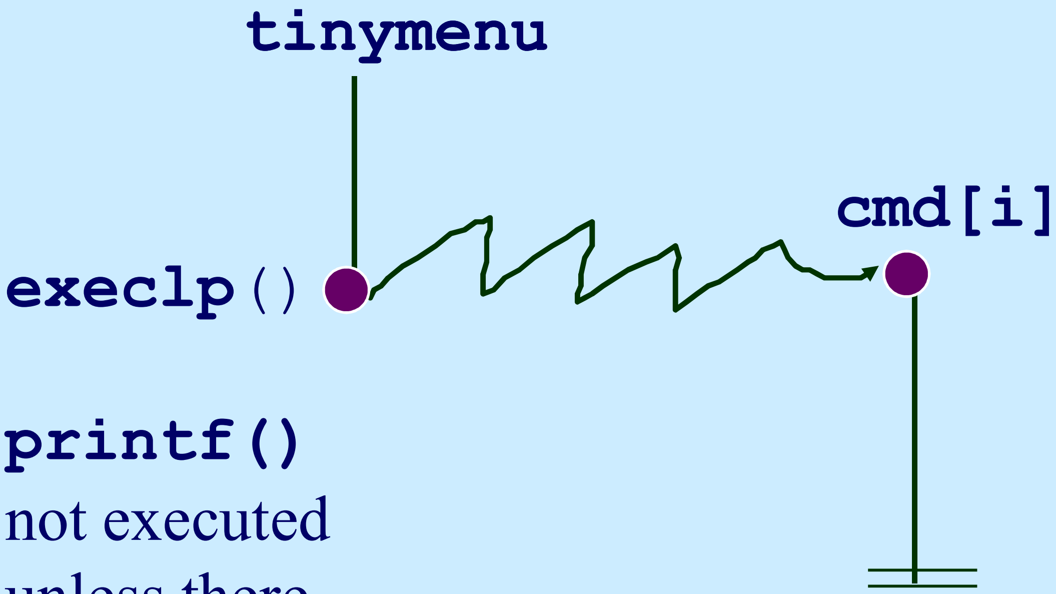
- if there’s a “v” in the function name, the code is expecting a vector (array) of string values. Again the array’s last element MUST be a NULL.

- if there’s an “e” in the function name, the function allows you to also pass along additional **ENV**ironment variables values the code may need to run.

tinymenu.c

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

void 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” );
  exit(1);
}
```

Execution



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

execl example

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

void main()
        {
        printf(“executing ls\n”);

        execl( “/bin/ls”, “ls”, “-1”, (char *)0 );

        /* if execl returns, the call has failed … */
        printf( “execl failed\n” );
        exit(1);
        }
```

List of parameters finishes with a NULL value

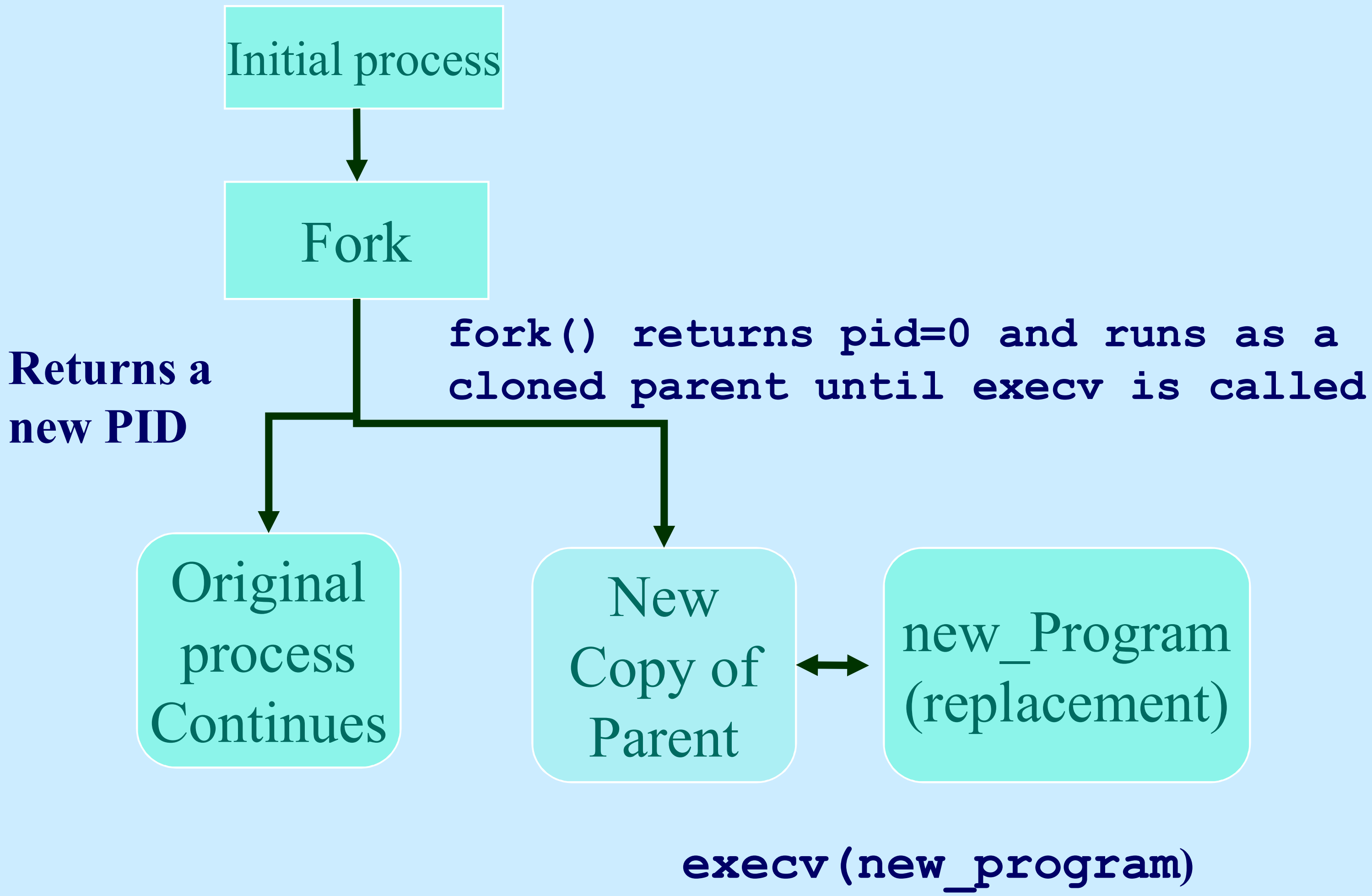
execv example

Array of parameters finishes with a NULL value

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

void main()
        {
        char *av[] = {“ls”, “-1”, (char *) 0};

        execv( “/bin/ls”, av );

        /* getting this far means an error … */
        printf( “execv failed\n” );
        exit(1);
        }
```

fork() and execv()

- `execv(new_program, argv[ ])`



Initial process

Fork

Returns a new PID

`fork() returns pid=0 and runs as a cloned parent until execv is called`

Original process Continues

New Copy of Parent

new_Program (replacement)

`execv(new_program)`

4. `wait()`

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

- Suspends calling process until child has finished. Returns the process ID of the terminated child if ok, -1 on error. This means the calling process is not using CPU as the scheduler ignores it.

- Used to synchronize process execution

- **`statloc`** is a pointer to an integer variable which will contain the status info. about the child (or can be `(int *)0`)

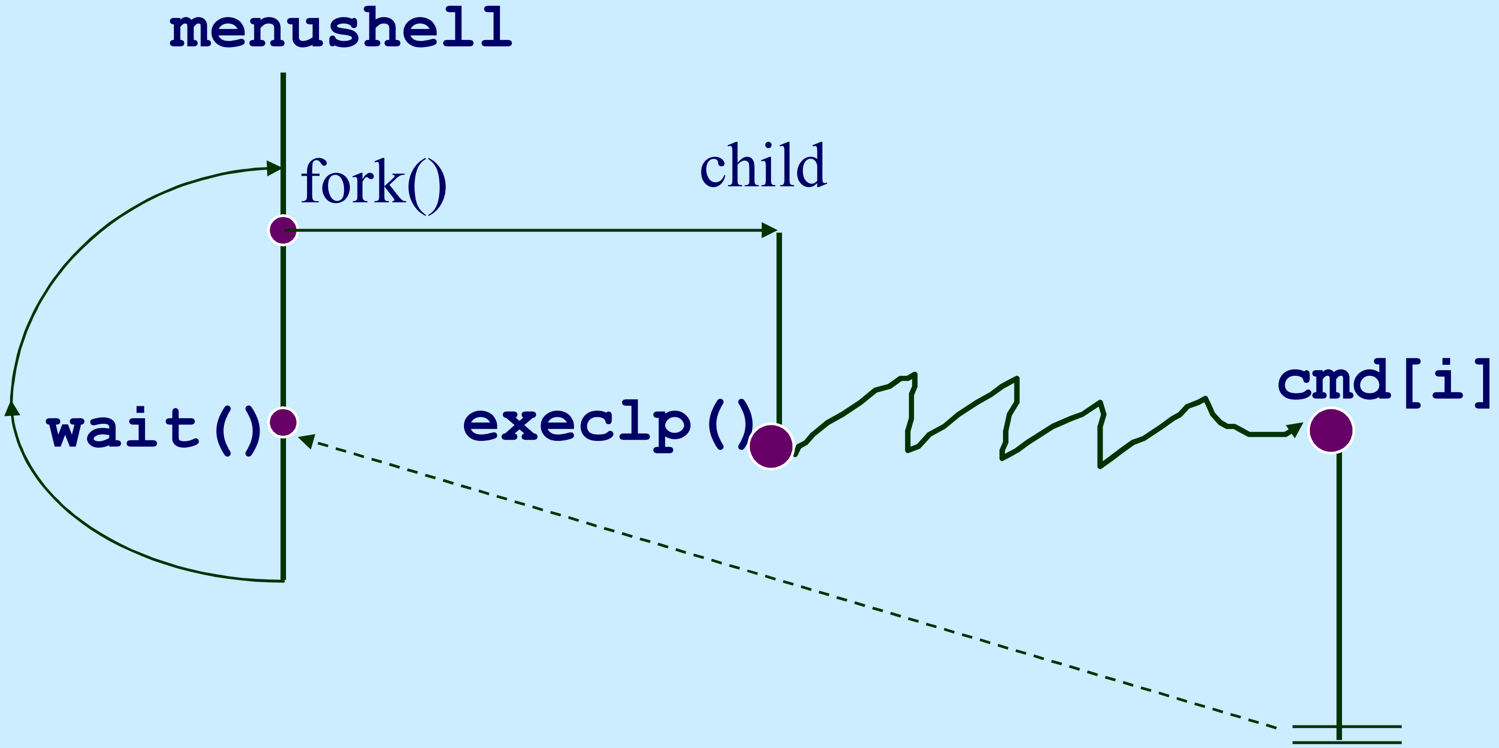
`wait()` Actions

- A process that calls `wait()` can:
  - *suspend* (block) if all of its children are still running, or
  - *return* immediately with the *termination* status of **a** child, or
  - *return* immediately with an *error* if there are no child processes.

menushell.c

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

void main()
{
     char *cmd[] = {“who”, “ls”, “date”};
     int i, done = 0;
     while( !done )
      {
              printf(“0=who 1=ls 2=date 3=terminate: ” );
              scanf( “%d”, &i );
              if(i != 3)
                       {
                       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” );
                               }
                       }
              else
                       done = 1;
     } /* while */
} /* main */
```

Execution



Macros for wait(status)

- WIFEXITED(*status*)
  - Returns true if the child exited normally.
- WEXITSTATUS(*status*)
  - Evaluates to *the least significant eight bits* of the return code of the child which terminated, which may have been set as the argument to a call to exit( ) or as the argument for a return.
  - This macro can only be evaluated if WIFEXITED returned non-zero.
- WIFSIGNALED(*status*)
  - Returns true if the child process exited *because of a signal* which was not caught.
- WTERMSIG(*status*)
  - Returns *the signal number* that caused the child process to terminate.
  - This macro can only be evaluated if WIFSIGNALED returned non-zero.

waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid( pid_t pid, int *status, int opts )
```

- waitpid - can wait for a particular child
- *pid* > 0
  - Wait for a specific child process with a process id of *pid*
- *pid* == -1
  - Wait for any child process.
  - Same behavior which wait( ) exhibits.
- *pid == 0*
  - Wait for any child in the process group of the current process
- *pid* < -1
  - Wait for any process in the process group (given by the absolute value of *pid*)

- *options*
  - Zero or more of the following constants can be ORed.
    - WNOHANG
      - Return immediately if no child has exited (allows the programmer to wait in a loop monitoring a situation but not blocking)
    - WUNTRACED
      - Returns the status for children which are stopped, and whose status has not been reported (because of signal).
- Return value
  - The process ID of the child which exited.
    - -1 on error;
    - 0 if WNOHANG was used and no child was available.

waitpid example

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

int main()
    {
   pid_t pid;
   int status, exit_status;

   if( (pid = fork()) < 0 )
          { perror(“fork failed”);
          exit(1); }

    if(pid == 0) /* child */
          {
          printf(“Child is sleeping … %d\n”, getpid());
          sleep(4);
          exit(5);
          }

/* parent code from here down */

    while(waitpid(pid, &status, WNOHANG) == 0)
          {/* getting this far means it is the parent */
          printf(“Still waiting … \n”);
          sleep(1);
          }
    if(WIFEXITED(status))
          {
          exit_status = WEXITSTATUS(status);
          printf(“Exit status from %d was %d\n”, pid, exit_status);
          }
          exit(0);

    } /* main */
```

OUTPUT

```
Child is sleeping ...
Still waiting ...
Still waiting ...
Still waiting ...
Still waiting ...
Exit status from 23320 was 5
```

Macros for waitpid

- WIFSTOPPED(*status*)
  - Returns true if the child process which caused the return is *currently stopped*.
  - This is only possible if the call was done using WUNTRACED.
- WSTOPSIG(status)
  - Returns *the signal number* which caused the child to stop.
  - This macro can only be evaluated if WIFSTOPPED returned non-zero.

Another Example: waitpid

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

int main(void)
   {
   pid_t pid;
   int status, done=0;

   if( (pid = fork() ) == 0 )
        { /* child */
        printf(“I am a child with pid = %d\n”, getpid());
        sleep(3);
        printf(“child terminates\n”);
        exit(0);
        }
```

Why?

```
     else
        { /* parent */
        while (!done)
                {
                printf(“Checking …\n”);
                waitpid( pid, &status, WUNTRACED );
                if( WIFSTOPPED(status) )
                        printf(“child stopped, signal(%d)\n”,
                            WSTOPSIG(status));
                else if( WIFEXITED(status) )
                        {
                        printf(“normal termination with status(%d)\n”,
                            WEXITSTATUS(status));
                        done = 1;
                        }
                else if (WIFSIGNALED(status))
                        {
                        printf(“abnormal termination, signal(%d)\n”,
                            WTERMSIG(status));
                        done = 1;
                        }

                } /* while */
        } /* parent */
        exit(0);
} /* main */
```

5. 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.

Example

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

int globvar = 6;
char buf[] = “stdout write\n”;

int main(void)
      {
      int w = 88;
      pid_t pid;
      write( 1, buf, sizeof(buf)-1 );
      printf( “Before fork()\n” );

      if( (pid = fork()) == 0 )
               {                  /* child */
               globvar++;
               w++;
               }
      else if( pid > 0 )          /* parent */
               sleep(2);
      else
               perror( “fork error” );

      printf( “pid = %d, globvar = %d, w = %d\n”, getpid(), globvar, w );
      return 0;

      } /* end main */
```

Output

❖
```
$ globex
stdout write     /* write not buffered */
Before fork()
pid = 430, globvar = 7, w = 89  /*child */
pid = 429, globvar = 6, w = 88 /* parent */
```

6. Process File Descriptors

- A child and parent have copies of the file descriptors, but the R-W pointer is maintained at the O/S level (by the kernel):
  - the R-W pointer is **shared**

- This means that a `read()` or `write()` in one process (child or parent) will affect the other process since the R-W pointer is changed.

Example: File used across processes

```
void printpos( char *msg, int fd )
                   /* Print position in file */
{
  long int pos;

  if( (pos = lseek( fd, 0, SEEK_CUR) ) < 0 )
          perror(“lseek”);

  printf( “%s: %ld\n”, msg, pos );
}
```

Example: File used across processes

```
void printpos(char *msg, int fd);

int main(void)
{
   int fd;                /* file descriptor */
   pid_t pid;
   char buf[10];          /* for file data */
              :
   if ((fd=open(“data-file”, 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” );

     } /* main */
```

Output

```
$ shfile

Before fork: 10
Child before read: 10
Child after read: 20
Parent after wait: 20
```

what's happened?

8. Special Exit Cases

Two special cases:

- 1) A child exits when its parent is not currently executing `wait()`
  - the child becomes a *zombie*
  - `status` data about the child is stored until the parent does a `wait()`
- 2) A parent exits when 1 or more children are still running
  - children are adopted by the system's initialization process (`/etc/init`)
    - it can then monitor/kill them

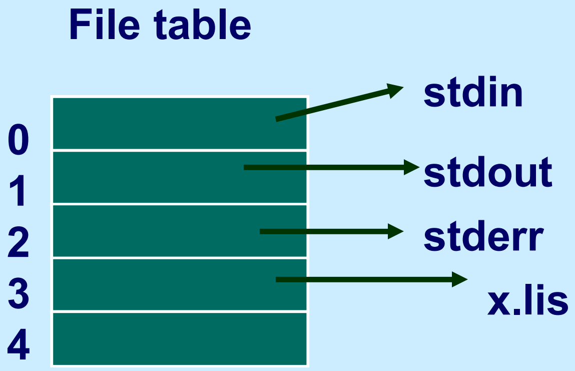
9. I/O redirection

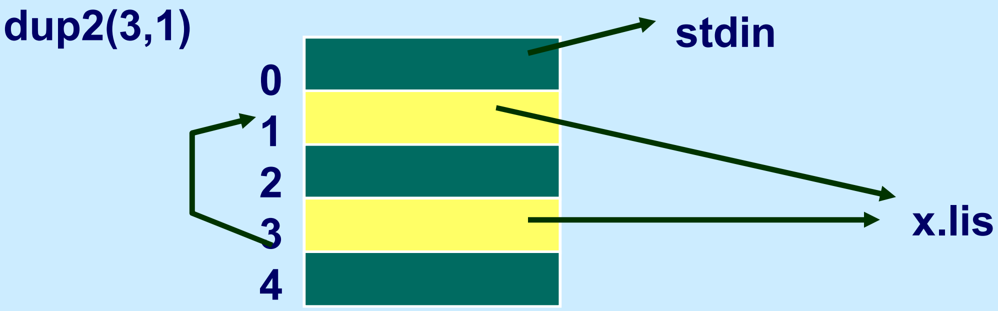
- The trick: you can change where the standard I/O streams are going/coming from after the fork but before the exec

Redirection of standard output

- Example implement shell: ls > x.ls
- program:
  - Open a new file x.lis
  - Redirect standard output to x.lis using **dup** command
    - everything sent to standard output is now sent to x.lis
  - execute ls in the process

- dup2(int fin, int fout) - copies fin to fout in the file table





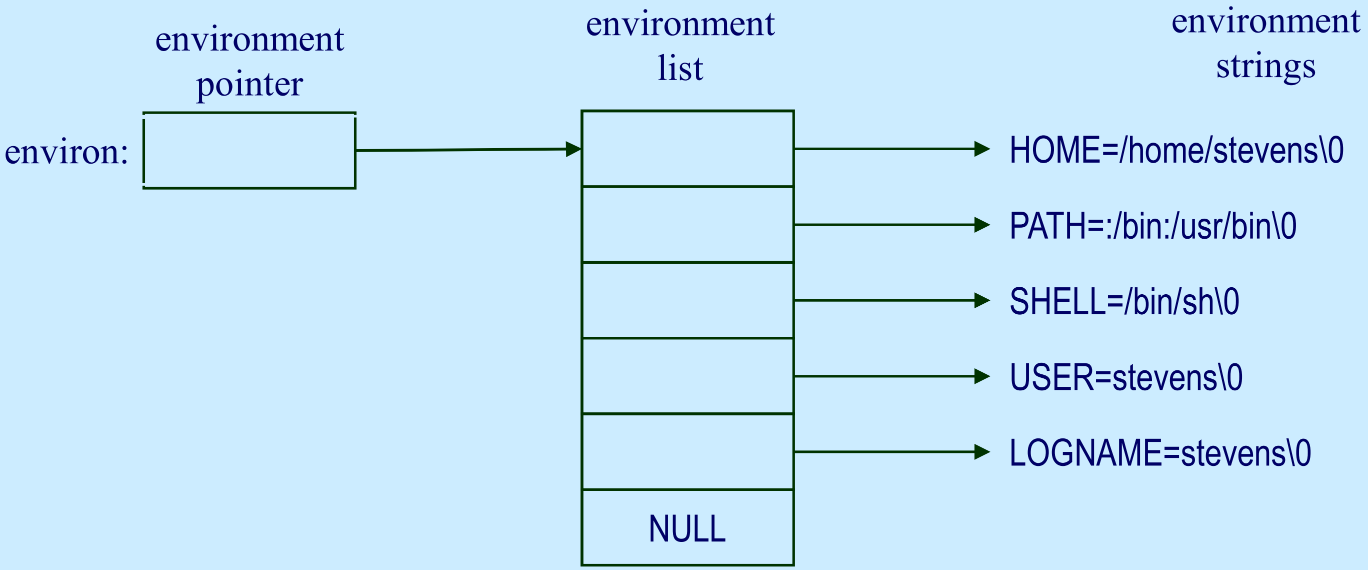
Example - implement ls > x.lis

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

int main ()
        {
        int fileId;
        fileId = creat( "x.lis",0640 );
        if( fileId < 0 )
                {
                printf("error creating x.lis\n" );
                exit (1);
                }
        dup2( fileId, 1 ); /* copy fileID to stdout */

        execl( "/bin/ls", "ls", "-lt" , (char *) 0 );
}
```

11. Environment

- extern char **environ;

  int main(int *argc*, char **argv[ ]*, char **envp[ ]*)



Example: environ

```
#include <stdio.h>

void main( int argc, char *argv[], char *envp[] )
   {
   int i;
   extern char **environ;

   printf( “from argument envp\n” );

   for( i = 0; envp[i]; i++ )
       puts( envp[i] );

   printf(“\nFrom global variable environ\n”);

   for( i = 0; environ[i]; i++ )
          puts(environ[i]);
}
```

getenv

- #include <stdlib.h>

  char *getenv(const char **name*);

  – Searches the environment list for a string that matches the string pointed to by *name*.

  – Returns a pointer to the value in the environment, or NULL if there is no match.

putenv

- #include <stdlib.h>
  int putenv(const char **string*);
  - Adds or changes the value of environment variables.
  - The argument *string* is of the form name=value.
  - If name does not already exist in the environment, then string is added to the environment.
  - If name does exist, then the value of name in the environment is changed to value.
  - Returns zero on success, or -1 if an error occurs.

Example : getenv, putenv

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

void main(void)
{
    printf(“Home directory is %s\n”, getenv(“HOME”));

    putenv(“HOME=/”);

    printf(“New home directory is %s\n”, getenv(“HOME”));
}
```

```
[193][rhurley@tyr:~]$ a.out
Home directory is /home/rhurley
New home directory is /
```

Pthreads

- fork() is not the only way to have your process create new “processes”.
- There are light-weight processes called threads.
- You can create these very easily but they do come with some extra complexities.

- Threads are lightweight processes as they share most of everything with the parent process.
- This includes all of the variables and the process ID

pthreads (POSIX threads)

In the simplest description, pthreads allow you to run “functions” as independent “processes”.

They will run in parallel to your original code.

If you create multiple of them, they will also run in parallel

REMEMBER: unlike fork(), threads share the same memory space.

You must take precautions to ensure that one thread doesn’t clobber a variable being used by a different thread!

pthreads - usage

Not unlike wait() and waitpid(), it is proper to have the main program wait for all threads to terminate before exiting.

For each thread:

- create thread
- join the thread (equivalent to a wait())

Your main program can go on doing all sorts of things in parallel while the threads are running.

pthreads – the complexity

pthreads are conceptually easy. It may take some time however to wrap your head around passing values to the thread and returning exit status values.

The multiple levels of “casting” data types can be quite daunting.

It is often recommended to use structures for passing information.

Remember that any variables declared inside the function disappear BEFORE you can return them to the main routine.

pthread_create()

pthreads (POSIX threads) are created using this system call:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg);
```

Where:

thread: pointer to a long unsigned integer to hold the thread number

attr: are the thread attributes (typically just NULL)

start: the name of the function we want the thread to run.

arg: A pointer to a structure or a variable that we want to pass to the function. NOTICE: THERE’S ONLY ONE!

pthread_create:

e.g.

```
status = pthread_create(&t1, NULL, threadLoop, (void *)&snooze[0]);
```

Here, t1 holds the thread number, we pass no extra attributes, the function we want the thread to run is called threadLoop and we are passing to it, the pointer to the 0th element of the array snooze.

pthread_join

int pthread_join(pthread_t thread, void **retval);

thread: value is that returned from pthread_create

retval: the exist status value from the function we called.,

A quick example

Here is a quick little piece of code.

It essentially creates 3 threads

each thread receives a “snooze” value

Each thread will loop and increment a counter by the snooze value

The thread then goes to sleep for “snooze” seconds.

```
//
//   Globals
//
   int a_ptr[3]  = {0, 1, 2};
   int snooze[3] = {1,11, 7};
   int sums[3]   = {0, 0, 0};

static void * threadLoop(void *sleep_ptr)
{
   char  all_tabs[3]="\t\t\t";
   char  prefix[3];

   int loop_ptr;
   int ptr;
   int counter;
   int thread_snooze;

   pthread_t thread_number;

   ptr = *((int *) sleep_ptr);
   thread_snooze = snooze[ptr];
   thread_number = pthread_self();

   printf("In thread %d. [TID=%d] Sleep delay=%d\n",ptr,thread_number,thread_snooze);
   strncpy(prefix,all_tabs,ptr);

   counter = 0;
   for ( loop_ptr=0; loop_ptr<10; loop_ptr++)
     {
       printf("%s[thread=%d] [%d]\n",prefix,ptr,counter);
       counter += thread_snooze;
       sleep(thread_snooze);
     }
    sums[ptr] = counter;

}
```

```c
int main(int argc, char *argv[])
{
   pthread_t t1,t2,t3;
   void *res;
   int s;

   int snooze[3] = {1,11,7};
   int sums[3] = {0, 0, 0};

   s = pthread_create(&t1, NULL, threadLoop, (void *)&snooze[0]);
   if (s != 0)       printf("Error creating thread 1\n");
   s = pthread_create(&t2, NULL, threadLoop, (void *)&snooze[1]);
   if (s != 0)      printf("Error creating thread 2\n");
   s = pthread_create(&t3, NULL, threadLoop, (void *)&snooze[2]);
   if (s != 0)      printf("Error creating thread 3\n");

   printf("Message from main()\n");

   s = pthread_join(t1, &res);
   if (s != 0)        printf("Unable to join the finished thread 1\n");
   printf("Thread %d returned %d\n", t1, res);
   s = pthread_join(t2, &res);
   if (s != 0)        printf("Unable to join the finished thread 2\n");
   printf("Thread %d returned %d\n", t2, res);
   s = pthread_join(t3, &res);
   if (s != 0)        printf("Unable to join the finished thread 3\n");
   printf("Thread %d returned %d\n", t3, res);

   exit(EXIT_SUCCESS);
}
```

Running the code:

```
[jacques@loki 3380]$ gcc -pthread -o ptla pthread_loop_a.c
[jacques@loki 3380]$ ./ptla
In thread 0. [TID=943265536] Sleep delay=1
Message from main()
In thread 1. [TID=934872832] Sleep delay=11
        [thread=1] [0]
In thread 2. [TID=926480128] Sleep delay=7
                [thread=2] [0]
[thread=0] [0]
[thread=0] [1]
…
[thread=0] [5]
[thread=0] [6]
                [thread=2] [7]
[thread=0] [7]
[thread=0] [8]
[thread=0] [9]
Thread 943265536 returned 0
        [thread=1] [11]
                [thread=2] [14]
                [thread=2] [21]
        [thread=1] [22]
                [thread=2] [28]
        [thread=1] [33]
                [thread=2] [35]
                [thread=2] [42]
        [thread=1] [44]
                [thread=2] [49]
        [thread=1] [55]
                [thread=2] [56]
                [thread=2] [63]
        [thread=1] [66]
…
        [thread=1] [99]
Thread 934872832 returned 1
Thread 926480128 returned 2
[jacques@loki 3380]$
```

Let’s look at the job ancestry!

When we run this code, we can look for the individual threads:

```
[jacques@loki 3380]$ ./process_tree.pl ptla
Looking for the ancestry of the process: ptla

root       919     1  0 Feb07 ?        00:00:55 /usr/sbin/sshd -D
root     27704   919  0 Feb11 ?        00:00:00  \_ sshd: jacques [priv]
jacques  27720 27704  0 Feb11 ?        00:00:03  |   \_ sshd: jacques@pts/0
jacques  27721 27720  0 Feb11 pts/0    00:00:00  |       \_ -bash
jacques   9876 27721  0 11:05 pts/0    00:00:00  |           \_ ./ptla

[jacques@loki 3380]$ ps -ef | grep -e ptl
jacques   9876 27721  0 11:05 pts/0    00:00:00 ./ptl
jacques   9898   533  0 11:05 pts/4    00:00:00 grep --color=auto -e ptla
[jacques@loki 3380]$
```

Unlike the fork(), pthreads do not appear as individual processes we can readily identify using “ps”.

The threads share the same process space as the parent “thread”

pthreads

The allow you to parallelize functions within your code.

Care must be taken however to ensure data integrity across all threads.

Because all threads share the same memory space, mutexes must be employed to help synchronize the execution of critical sections of code.