Process in unix

Create ; synchronize and exec commands

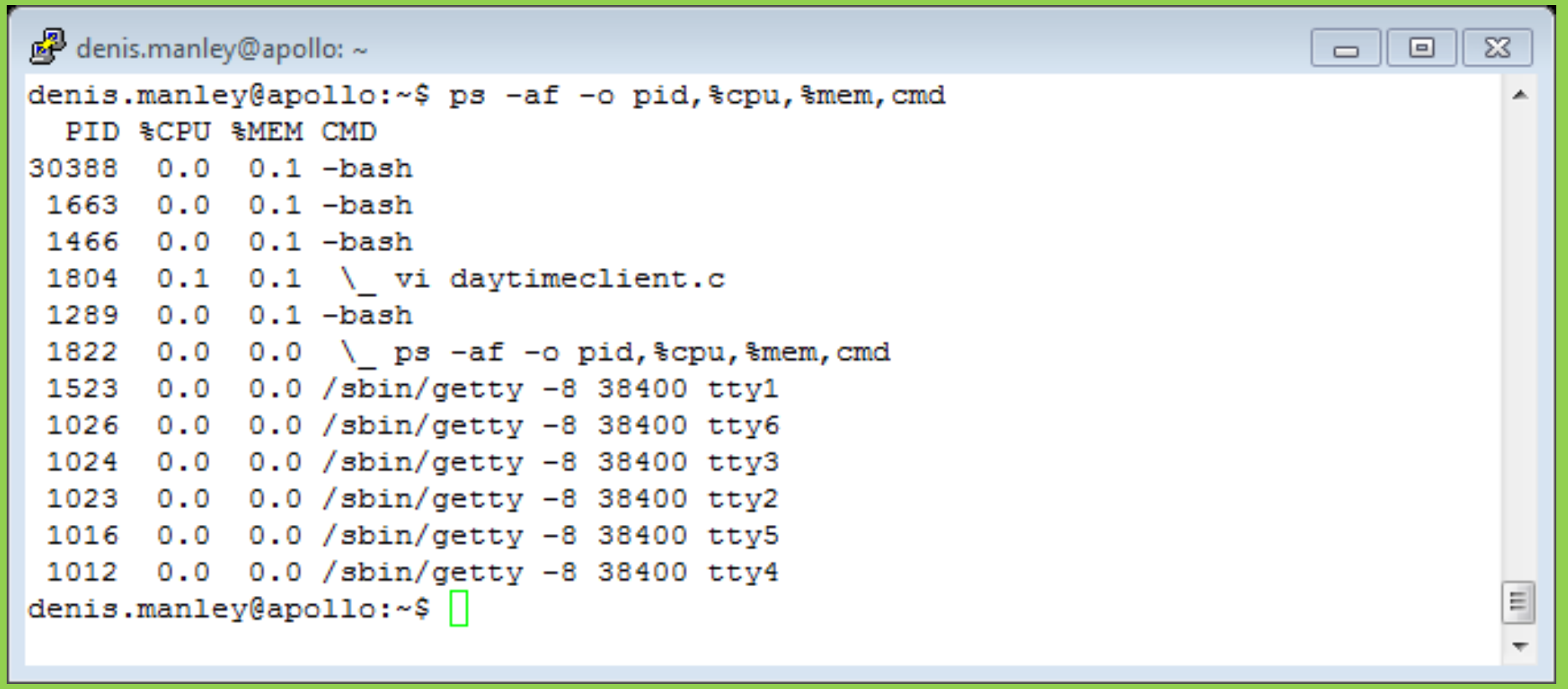
Process in Unix

- The ps command shows basic information about the PCBs
- **PID time DESCRIPTION (CMD)**
- e.g. the **man ps** shows other command line arugments; e.g.

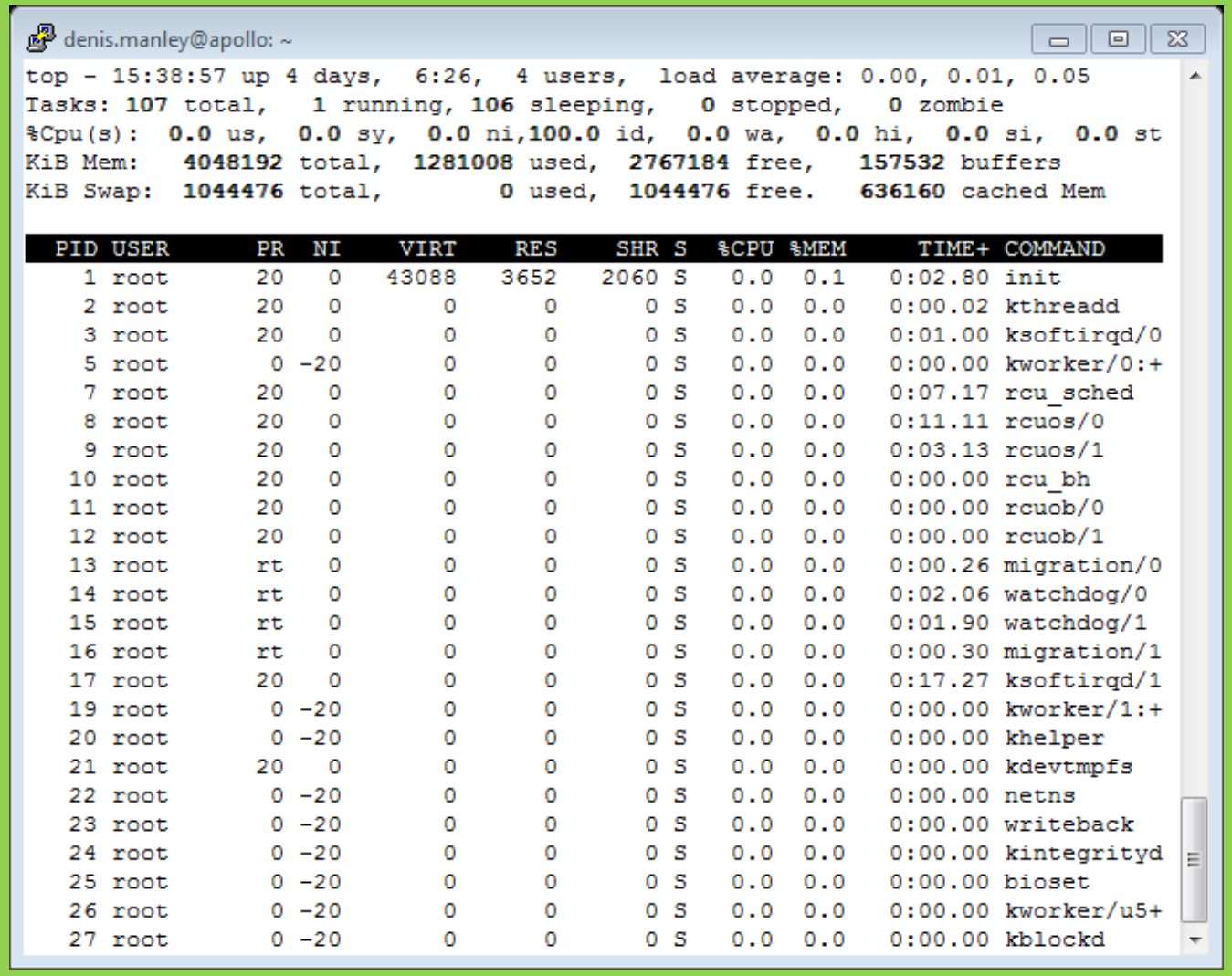
- %cpu %CPU cpu utilization of the process in "##.#" format.
- Currently, it is the CPU time used divided by the time the
- process has been running (cputime/realtime ratio),
- expressed as a percentage. It will not add up to 100%
- unless you are lucky. (alias pcpu).

- %mem %MEM ratio of the process's resident set size to the physical
- memory on the machine, expressed as a percentage.
- (alias pmem).

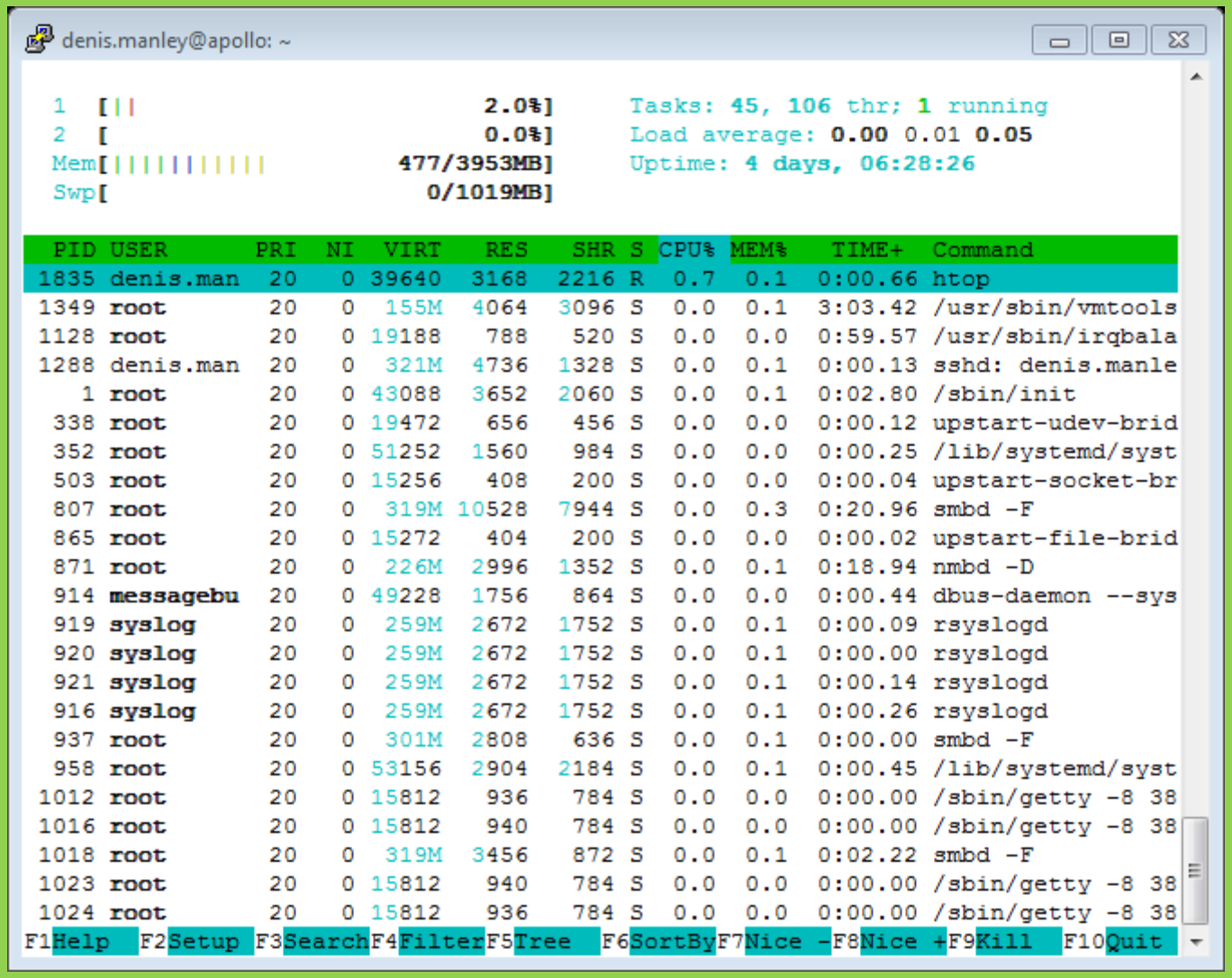
Example more detailed ps command



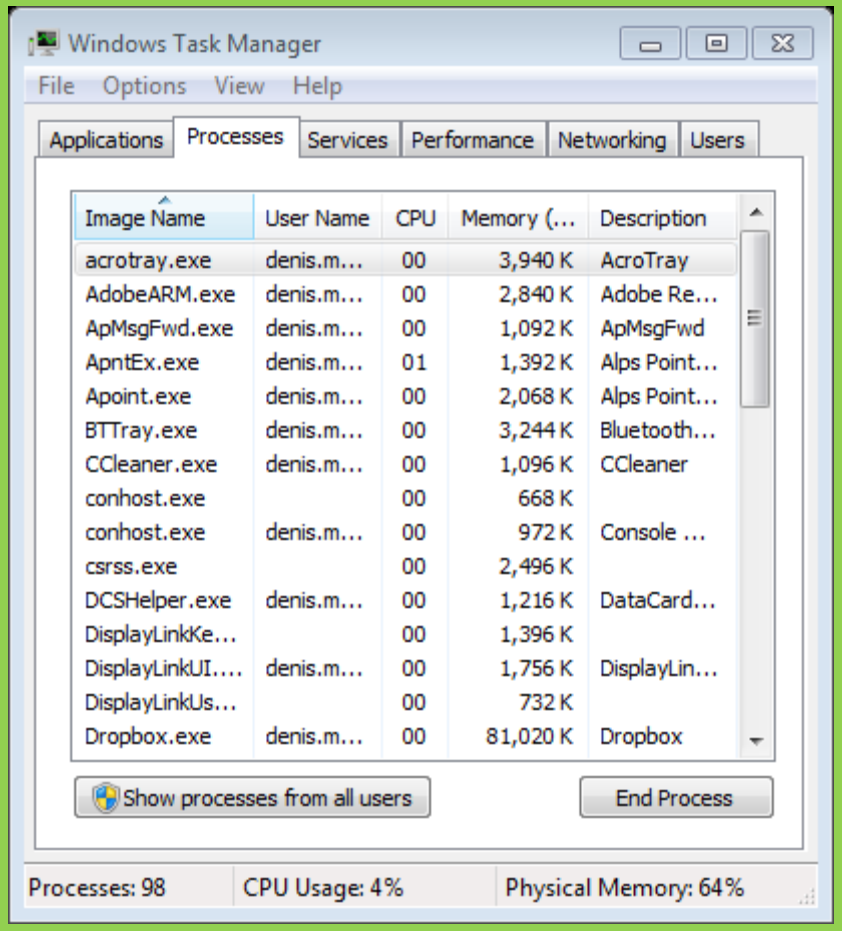
Top commands in linux: list of processes



htop command in linux



Process details in windows



Job and Process States

- Status changes: as a job or process moves through the system
  - HOLD and placed in a queue (sometimes a priority queue
  - READY (job begins processing if enough resources available)
  - WAITING (can not continue until specific resource available e.g. i/o request
  - RUNNING: processing
  - FINISHED

Process Creation: Unix

- In Unix, processes are created using fork()
- Creates and initializes a PCB and creates a new address space
- Initializes the address space with a copy of the entire contents of the address space of the parent
- Initializes the kernel resources to point to the resources used by parent (e.g., open files)
- Places the child PCB on the ready queue
- Fork returns twice: Returns the child's PID to the parent, "0" to the child

Process Creation

- Address space
  - Child duplicate of parent
  - Child has a program (process) loaded into it
- UNIX/Linux examples
  - The `fork()` system call in Unix creates a new process. After a successful `fork()` call, *two copies* of the original code will be running. In the original process (the parent) the return value of `fork()` will be the process ID of the child. In the new child process the return value of `fork()` will be 0

Synchronization

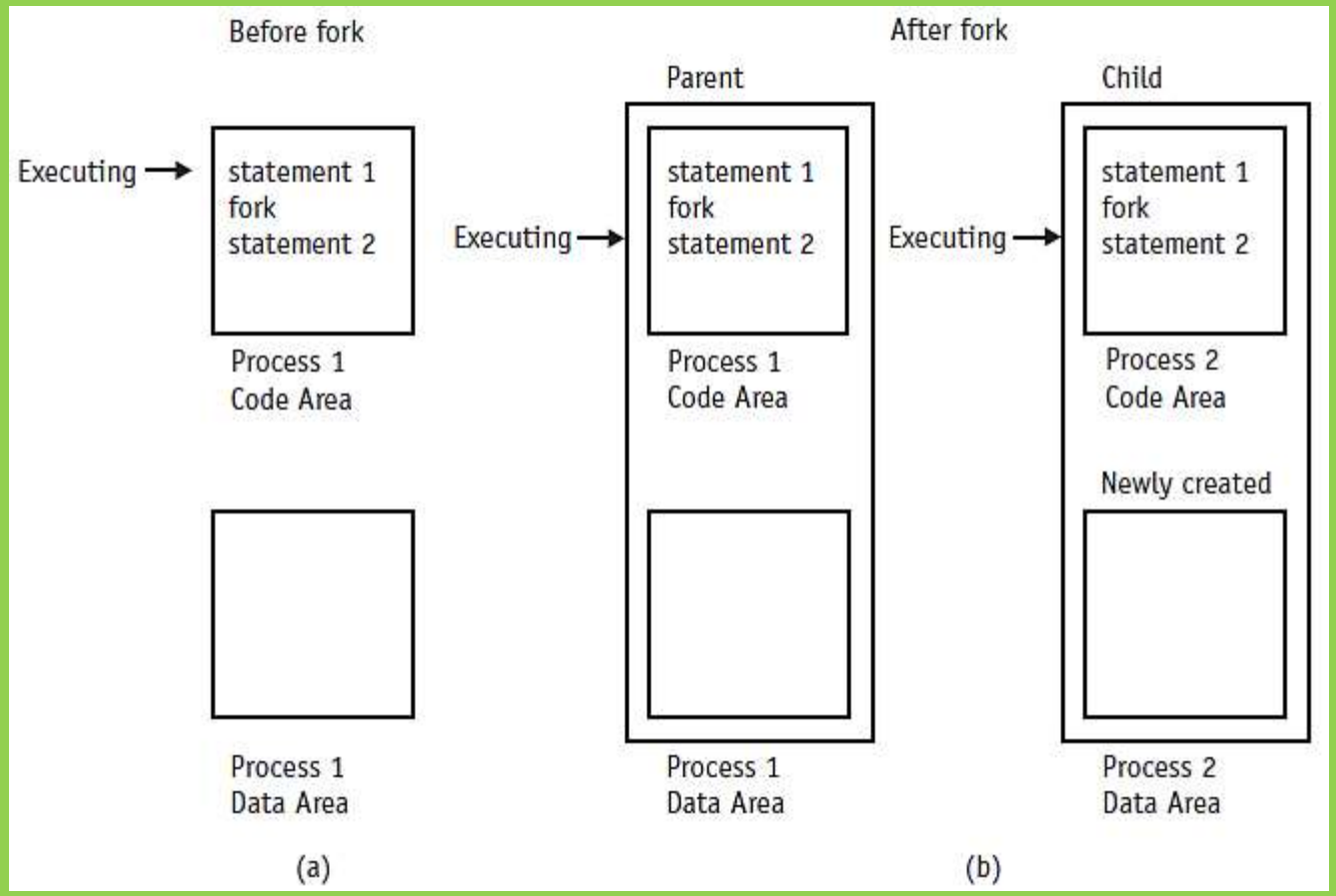
- **Fork()**
  - UNIX command: generates one program from another program
  - Gives second program all first program attributes
  - code, global data, heap and stack, registers (including *program counter*), and open files.
  - Saves first program in original form
  - Splits program: two versions of the parent (parent and child)
    - Both run from statement after `fork` command
  - `fork` executed
    - "Process id" (pid) generated
    - Ensures each process has unique ID number

Synchronization (cont'd.)

when the fork command is received, the parent process:

shown in (a) begets the child process shown

in (b) and Statement 2 (one of the child and one of the parent) is executed twice.



Cengage Learning 2014

Fork1.c main

denis.manley@apollo: ~/OS2/week6

```
/* program can be found at:


    demonstrates the return values of the
     fork function: return 0 for the child and pid of child to the parent

*/

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

#define   MAX_COUNT  20

void  ChildProcess(void);               /* child process prototype  */
void  ParentProcess(void);              /* parent process prototype */

void  main(void)
{
     pid_t  pid;

     pid = fork();  // fork creates a copy of the current process

     if (pid < 0)
        printf("error unable to fork process; ending program");

     if (pid == 0)
          ChildProcess();
     else
     {
         printf("the fork command returns the PID of the child: %d\n",pid);
         ParentProcess();
      }
}
                                                              1,1           Top
```

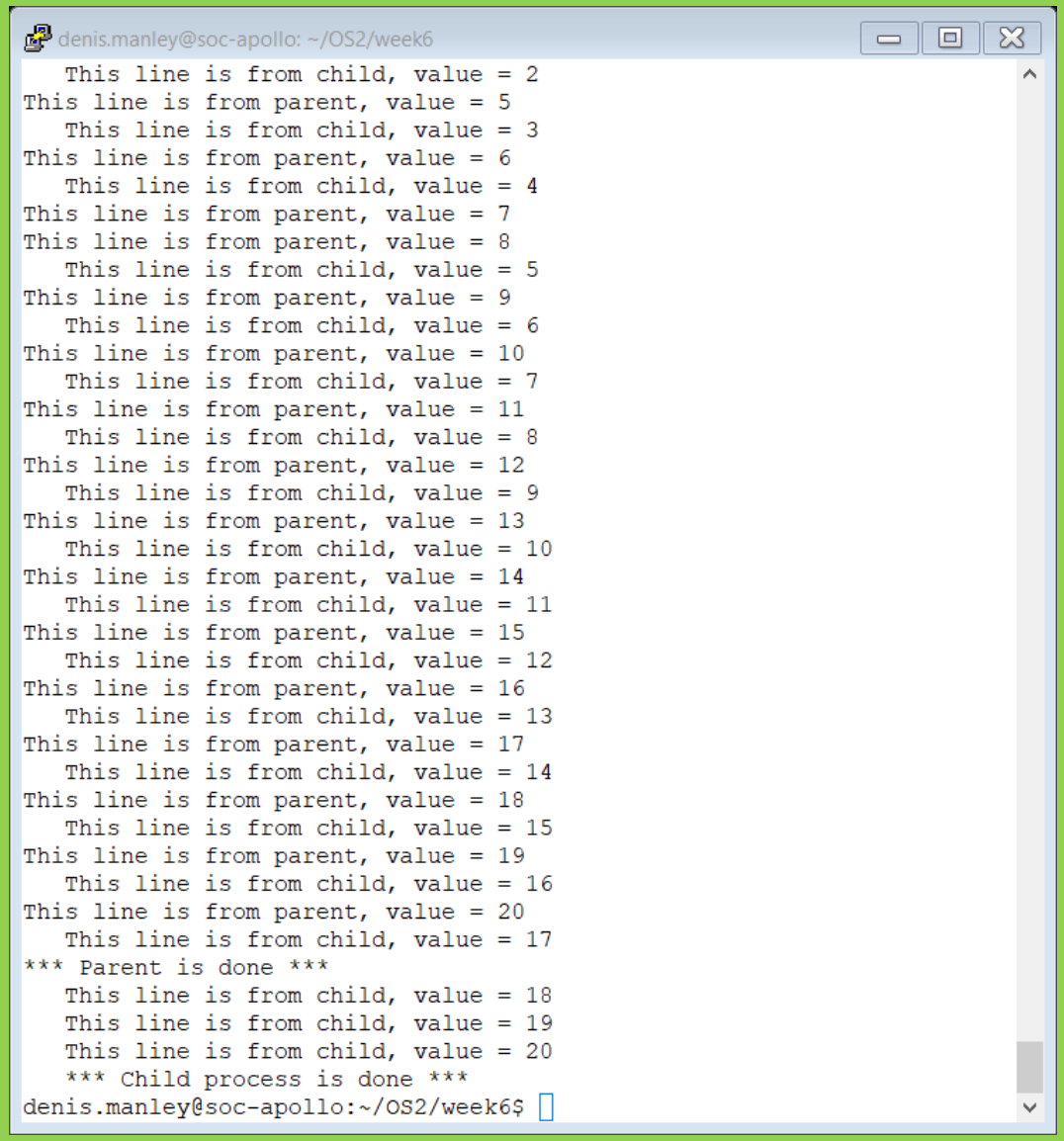
A simply program for fork: Fork1.c

denis.manley@apollo: ~/OS2/week6

```
void  ChildProcess(void)
{
     int   i,x, pid1, pid2;

     pid1 = getpid();     // get PID of the parent process

     for (i = 1; i <= MAX_COUNT; i++){
          printf("\tThis line is from child (pid number %d) the pid of the child
s parent is %d, value = %d\n",pid1,pid2, i);
          for (x=0; x<1000000000;x++);  // code to slow the process
     }
     printf("   *** Child process is done ***\n");
}

void  ParentProcess(void)
{
     int   i,x, pid1;

     pid1= getpid();

     for (i = 1; i <= MAX_COUNT; i++){
          printf("This line is from parent (Pid %d), value = %d\n",pid1, i);
          for(x=0; x<1000000000;x++); // code to slow process
    }
     printf("*** Parent is done ***\n");
}
```

61,0-1 Bot

A sample Output Fork1.c



Modify Fork1.c

- Reduce the number of iterations in the for loop: Fork1.c and Explain the effect.

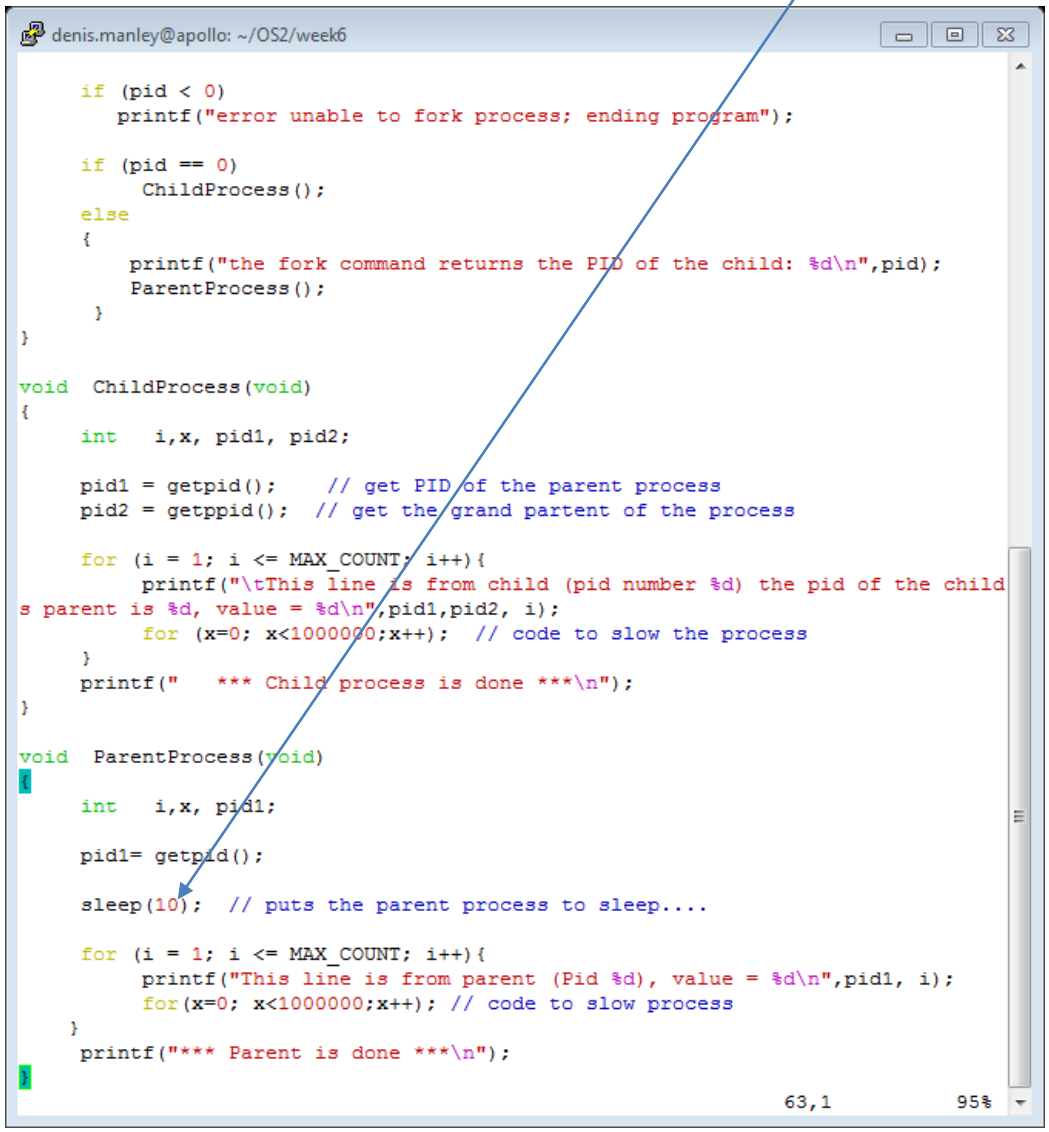
- Use ubuntu app/virtual machine and test the Fork1.c code: is it slower or faster?

Fork2.c main function

denis.manley@apollo: ~/OS2/week6

```
/*    fork2.c similiar to fork1.c but uses the sleep command to synchronise the parent and the child *
/


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

#define   MAX_COUNT  20

void  ChildProcess(void);                /* child process prototype  */
void  ParentProcess(void);               /* parent process prototype */

void  main(void)
{
     pid_t  pid;

     pid = fork();  // fork creates a copy of the current process

     if (pid < 0)
        printf("error unable to fork process; ending program");

     if (pid == 0)
          ChildProcess();
     else
     {
         printf("the fork command returns the PID of the child: %d\n",pid);
         ParentProcess();
      }
}
                                                                    4,0-1         Top
```

Fork2.c: Synchronise output with **sleep()** command:



Fork2.c: Synchronise output with sleep command:

```
denis.manley@apollo: ~/OS2/week6
the fork command returns the PID of the child: 1616
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 1
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 2
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 3
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 4
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 5
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 6
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 7
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 8
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 9
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 10
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 11
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 12
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 13
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 14
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 15
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 16
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 17
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 18
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 19
        This line is from child (pid number 1616) the pid of the childs parent is 1615, value = 20
   *** Child process is done ***
This line is from parent (Pid 1615), value = 1
This line is from parent (Pid 1615), value = 2
This line is from parent (Pid 1615), value = 3
This line is from parent (Pid 1615), value = 4
This line is from parent (Pid 1615), value = 5
This line is from parent (Pid 1615), value = 6
This line is from parent (Pid 1615), value = 7
This line is from parent (Pid 1615), value = 8
This line is from parent (Pid 1615), value = 9
This line is from parent (Pid 1615), value = 10
This line is from parent (Pid 1615), value = 11
This line is from parent (Pid 1615), value = 12
This line is from parent (Pid 1615), value = 13
This line is from parent (Pid 1615), value = 14
This line is from parent (Pid 1615), value = 15
This line is from parent (Pid 1615), value = 16
This line is from parent (Pid 1615), value = 17
This line is from parent (Pid 1615), value = 18
This line is from parent (Pid 1615), value = 19
This line is from parent (Pid 1615), value = 20
*** Parent is done ***
denis.manley@apollo:~/OS2/week6$
```

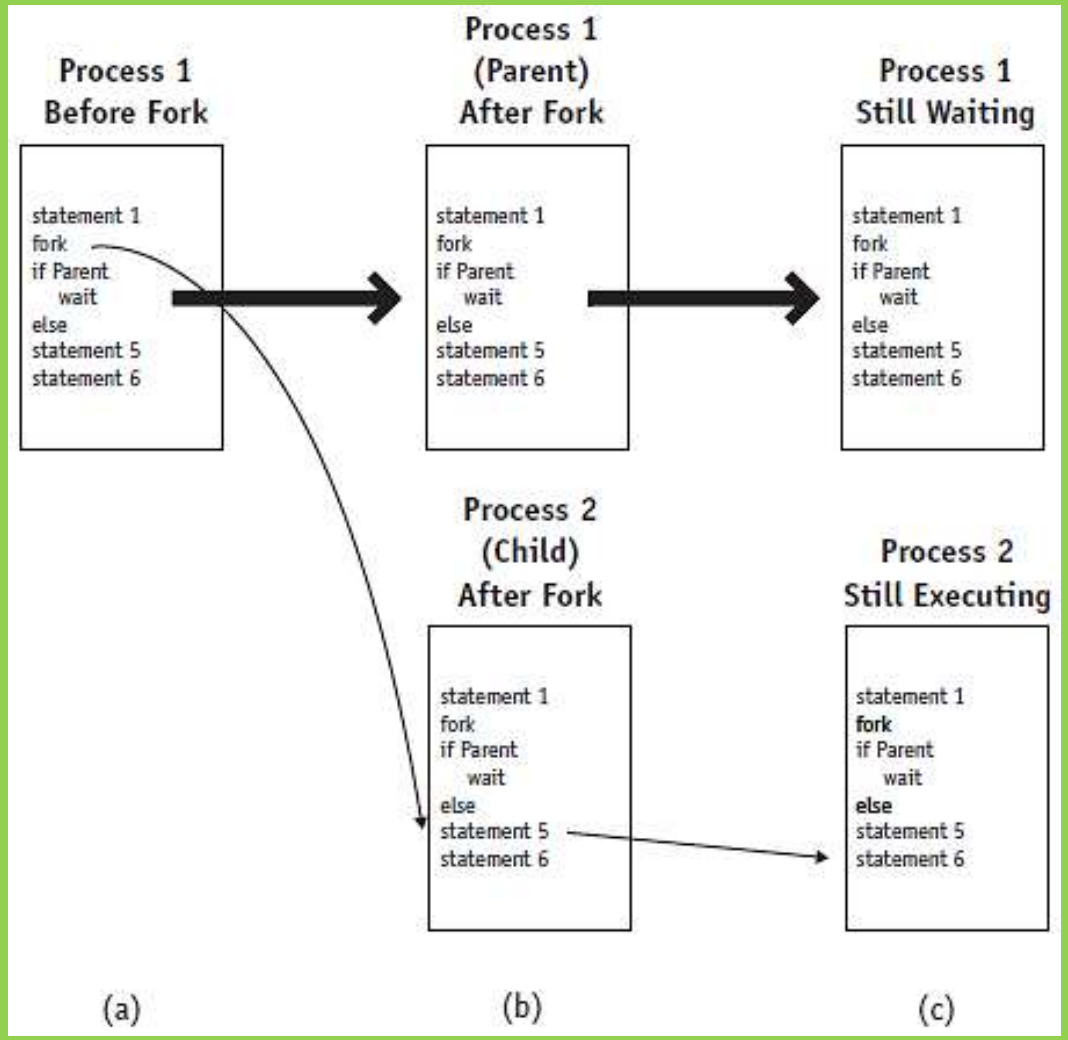
Change position of sleep command and observe the effect!!!!!

Synchronisation: Wait()

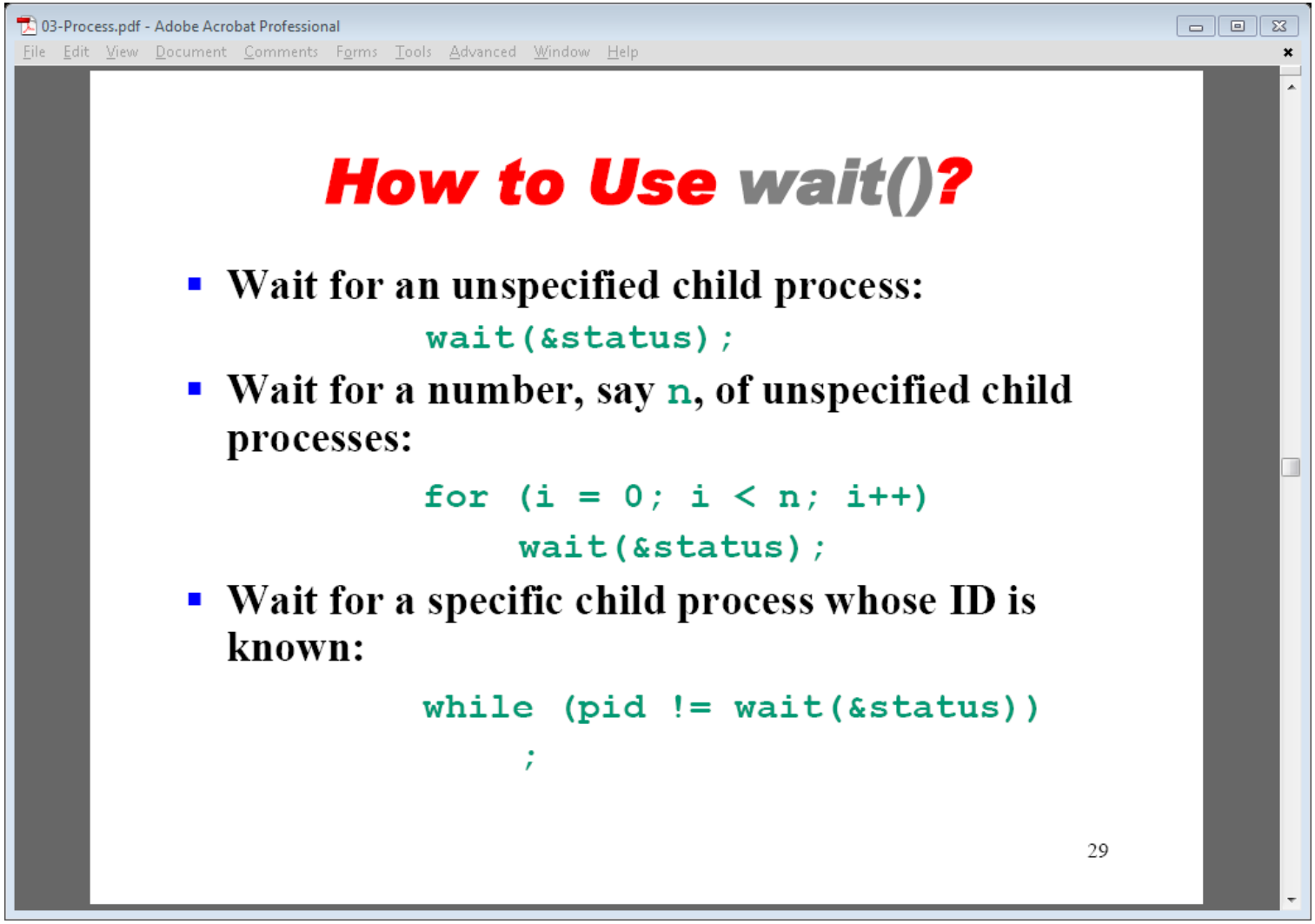
- **Wait()**
  - A better way to ensure “synchronisation is using the “wait()” command.
  - UNIX command: synchronizes process execution
    - Suspends parent until child finished
  - Program IF-THEN-ELSE structure
    - Controlled by pid value
    - pid > zero: parent process
    - pid = zero: child process
    - pid < zero: error in `fork` call
  - *Modify fork2.c program to use wait instead of sleep()*

The wait command

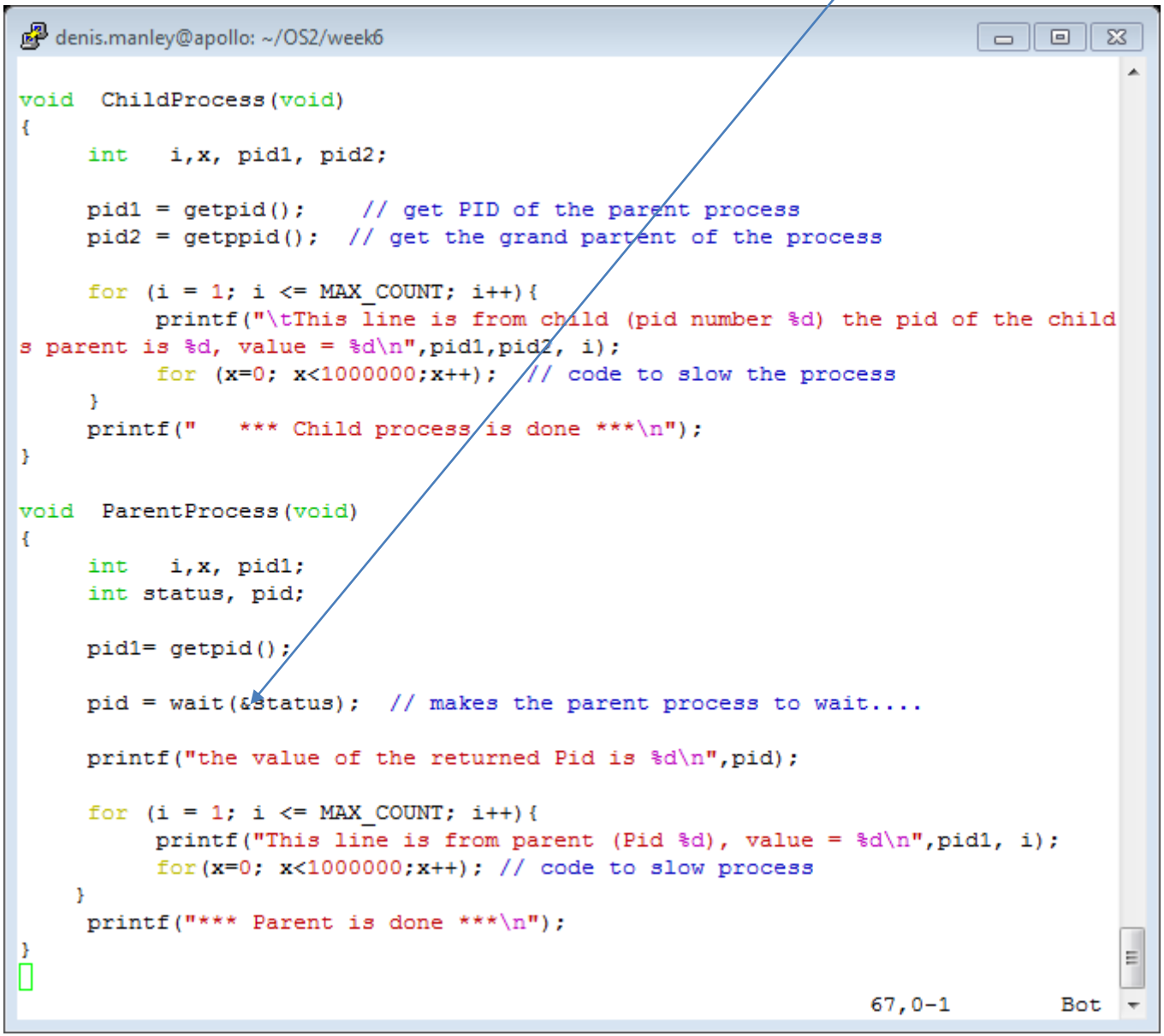
The wait command used in conjunction with the fork command will synchronize the parent and child processes. In (a) the parent process is shown before the fork, (b) shows the parent and child after the fork, and (c) shows the parent and child during the wait.



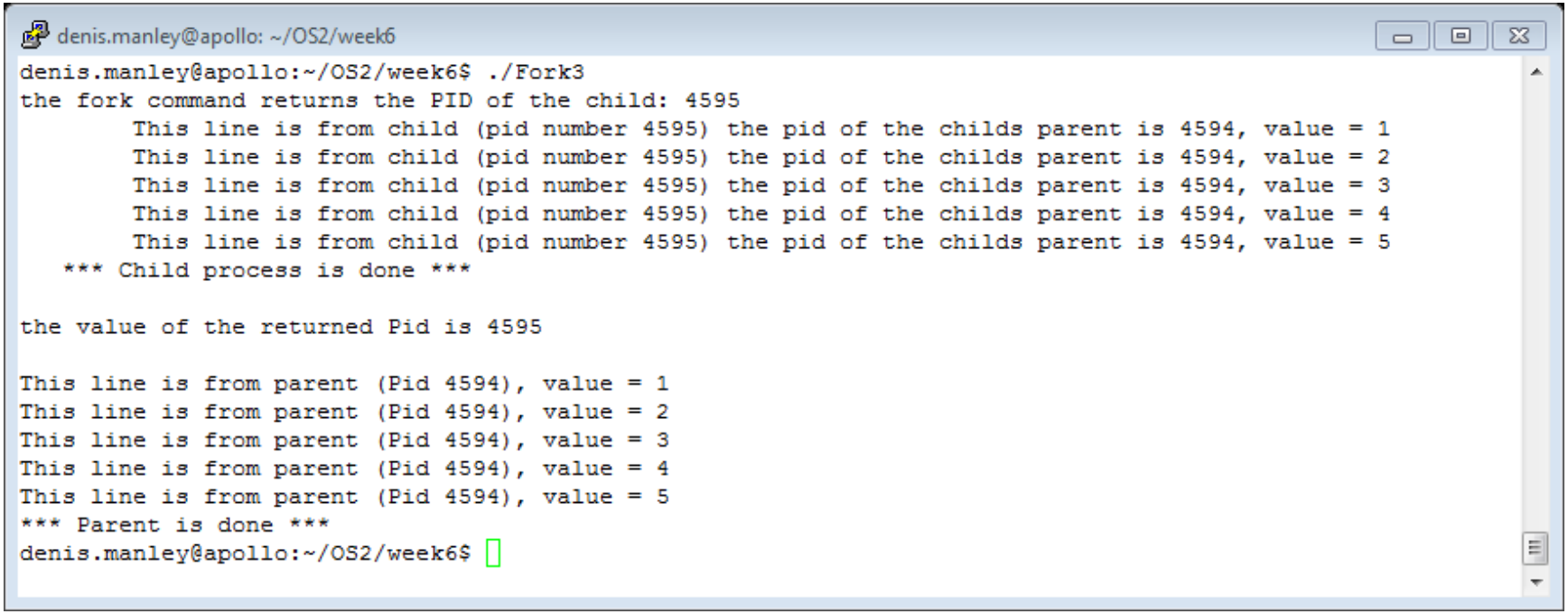
Using a wait()


03-Process.pdf - Adobe Acrobat Professional
File Edit View Document Comments Forms Tools Advanced Window Help
How to Use wait()?
Wait for an unspecified child process:
wait(&status);
Wait for a number, say n, of unspecified child processes:
for (i = 0; i < n; i++)
wait(&status);
Wait for a specific child process whose ID is known:
while (pid != wait(&status))
;
29

Sample code Fork3.c (Wait command)



Execution of Fork3.c (*MAX_COUNT = 5*)



int main(int argc, char **argv)

- argc is a variable that contains the number of command line arguments

- Argv is a character pointer array containing all the command line arguments.

- argv[0] is the first command line argument
- argv[1] is the second command line argument (if there is one…

- If the command line (after the prompt) is:
  - cp file1 file2
- what is the value returned by argv[2]?
- What value is returned by argc?

Pass command line arguments to c program: Fork4.c

denis.manley@apollo: ~/OS2/week6

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

int main(int argc, char **argv)
{
    printf("--beginning of program\n");

   if (argc != 3) {
        printf("number of command line arguments should be 3\n");
        printf("exiting program......\n");
        exit(0);
    }


    int counter = 0;
    int second;

    pid_t pid = fork();

    if (pid == 0)
    {
        // child process


        int max_c = atoi(argv[1]);

        printf("the second argument is %s\n",argv[1]);
        int i,x;
        for (i = 0; i < 5; ++i)
        {
            printf("child process: counter=%d\n", ++counter);
            for(x=0; x<max_c;x++);
        }
    }
    else if (pid > 0)
    {
        // parent process
        int j,x, status;

        int  max_p = atoi(argv[2]);

        printf("the third argument is %s\n",argv[2]);
        //pid = wait(&status);
        for (j=0; j < 5; ++j)
        {
            printf("parent process: counter=%d\n", ++counter);
            for (x=0;x<max_p;x++);
        }
    }
```

62,2 48%

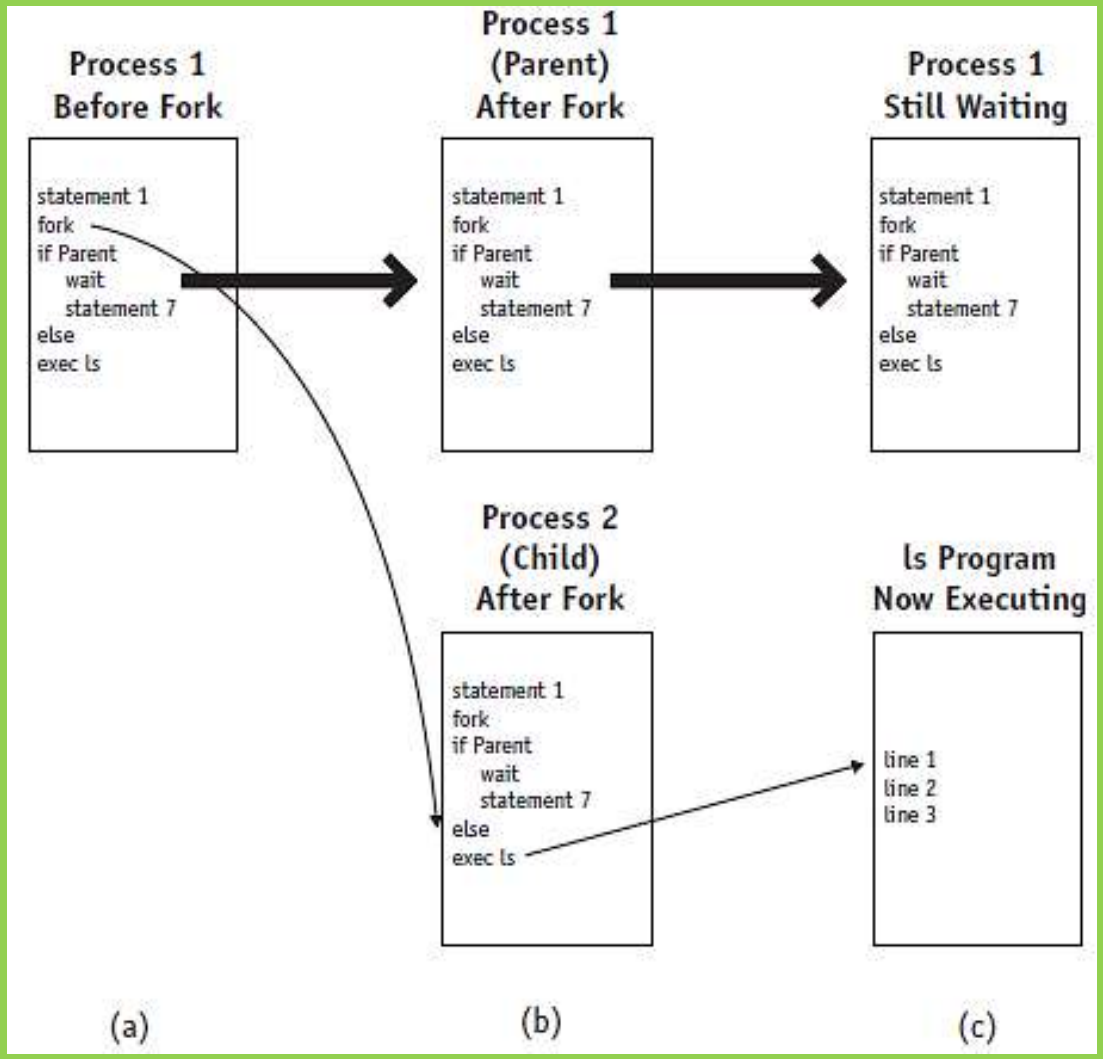
```
denis.manley@soc-apollo:~/OS2/week6$ ./fork4 1000 10000
--beginning of program
the third argument is 10000
parent process: counter=1
parent process: counter=2
the second argument is 1000
parent process: counter=3
child process: counter=1
child process: counter=2
child process: counter=3
child process: counter=4
child process: counter=5
--end of program--
parent process: counter=4
parent process: counter=5
--end of program--
denis.manley@soc-apollo:~/OS2/week6$ ./fork4 100000 1000000
--beginning of program
the third argument is 1000000
parent process: counter=1
the second argument is 100000
child process: counter=1
child process: counter=2
child process: counter=3
child process: counter=4
child process: counter=5
--end of program--
parent process: counter=2
parent process: counter=3
parent process: counter=4
parent process: counter=5
--end of program--
denis.manley@soc-apollo:~/OS2/week6$ ./fork4 10000000 100000000
--beginning of program
the third argument is 100000000
parent process: counter=1
the second argument is 10000000
child process: counter=1
child process: counter=2
child process: counter=3
child process: counter=4
child process: counter=5
--end of program--
parent process: counter=2
parent process: counter=3
parent process: counter=4
parent process: counter=5
--end of program--
denis.manley@soc-apollo:~/OS2/week6$
```


If not wait the process may interleave iterations. Try with the wait command

Process Creation: Unix

- How do we actually start a new program?
  - Use exec ((see *man exec* for more details)
  - Start new program execution from another program
- **E.g. int execvp(char *prog, char *argv[])**
  - Stops the current process (e.g. child of prompt)
  - Loads the program “prog” into the process’ address space
  - Initializes hardware context and args for the new program
  - Places the PCB of it onto the ready queue

Synchronization (cont'd.)

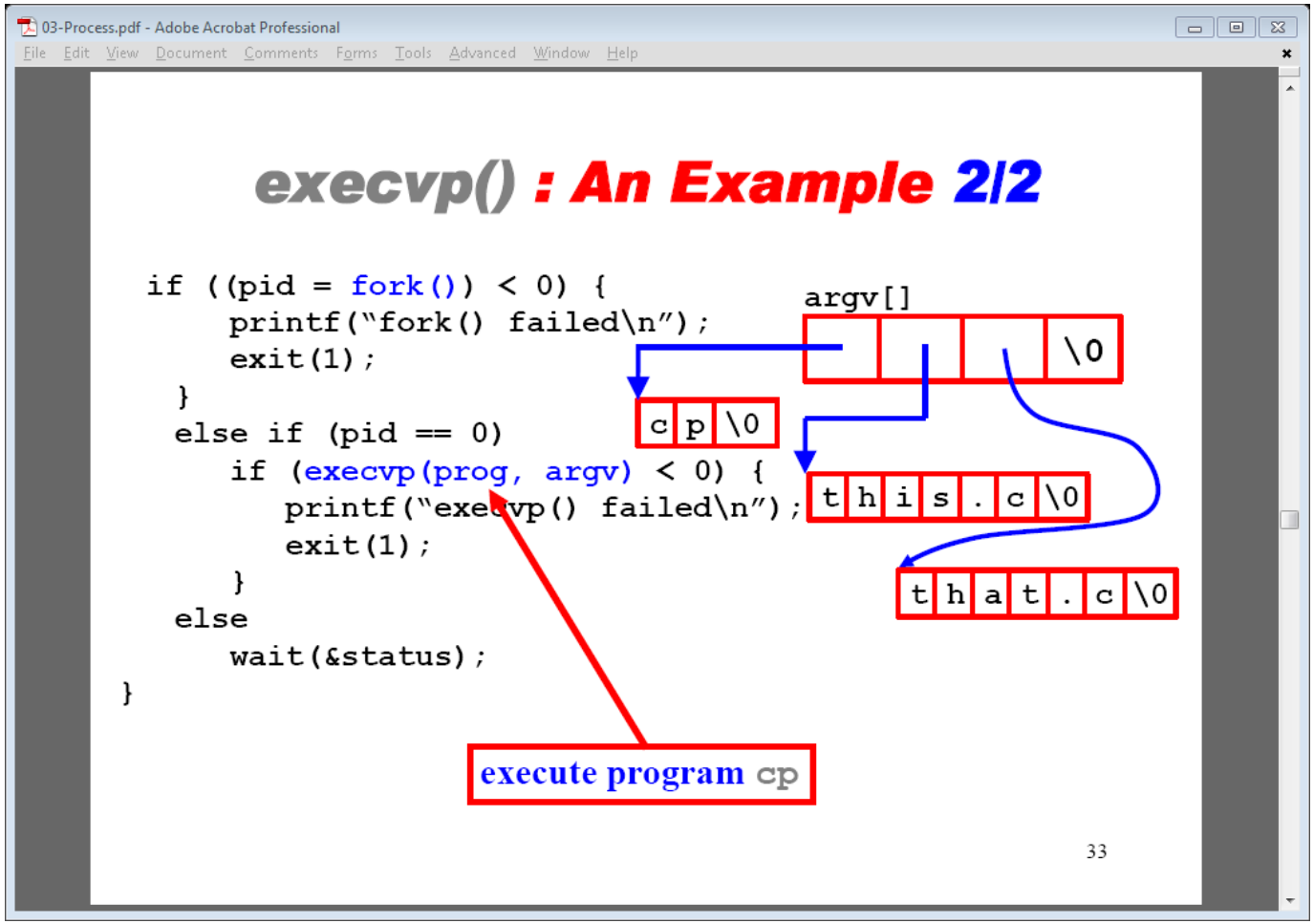


The exec command is used, mostly, after the fork and wait combination. In (a) the parent is shown before the fork, (b) shows the parent and child after the fork, and (c) shows how the child process (Process 2) is overlaid by the ls program after the exec command.

The exec commands

- `The exec()` system calls can be used after a `fork()` to replace the child process' memory space with a new program
- Successful `exec` call
  - Overlay second program over first
  - Only second program in memory
- No return from successful `exec` call
  - Parent-child concept: does not hold
- Create parent-child relationship
  - Call `fork`, `wait`, and `exec` in that order

Example with fork(), exec() and wait()



Refer to **execvp.c** program; This program simulates a **linux command line.**

Main () for execvp program

denis.manley@apollo: ~/OS2/week6

```
/* ------------------------------------------------------------------ */
/*                  The main program starts here                      */
/* ------------------------------------------------------------------ */

void  main(void)
{
     char  line[1024];             /* the input line                  */
     char  *argv[64];              /* the command line argument       */

     while (1) {                   /* repeat until done ....          */
          printf("Shell (enter exit to finish) -> ");     /*   display a prompt
            */
         fgets(line,1024,stdin);              /*   read in the command line
*/
//          printf("\ntest %s\n",line);
          parse(line, argv);       /*   parse the line                */
          if (strcmp(argv[0], "exit") == 0)  /* is it an "exit"?      */
          {     printf("exit program");
               exit(0);            /*   exit if it is                 */
          }
          execute(argv);           /* otherwise, execute the command */
     }

        printf("end main fnt\n");
}
```

111,0-1 Bot

Parse function

denis.manley@apollo: ~/OS2/week6

```
/* ------------------------------------------------------------------- */
/* PROGRAM  shell.c                                                    */
/*    This program reads in an input line, parses the input line       */
/* into tokens, and use execvp() to execute the command.               */

/*


        sample input ./exec  (run an exe file in current directory)
                                                                       */

/* ------------------------------------------------------------------- */

#include  <stdio.h>
#include <stdlib.h>
#include  <sys/types.h>

/* ------------------------------------------------------------------- */
/* FUNCTION  parse:                                                    */
/*    This function takes an input line and parse it into tokens.      */
/* It first replaces all white spaces with zeros until it hits a       */
/* non-white space character which indicates the beginning of an       */
/* argument.  It saves the address to argv[], and then skips all       */
/* non-white spaces which constitute the argument.                     */
/* ------------------------------------------------------------------- */

void  parse(char *line, char **argv)
{

//     printf("test parse..\n");

     while (*line != '\0') {       /* if not the end of line ..... */
          while (*line == ' ' || *line == '\t' || *line == '\n')
               *line++ = '\0';     /* replace white spaces */
          *argv++ = line;          /* save the argument position     */
          while (*line != '\0' && *line != ' ' &&
                 *line != '\t' && *line != '\n')
               line++;             /* skip the argument until ...    */
     }


         *argv = '\0';                 /* mark the end of argument list  */

//      printf("end parse\n");


}
                                                          1,1           Top
```

Execute function: equivalent to execvp()

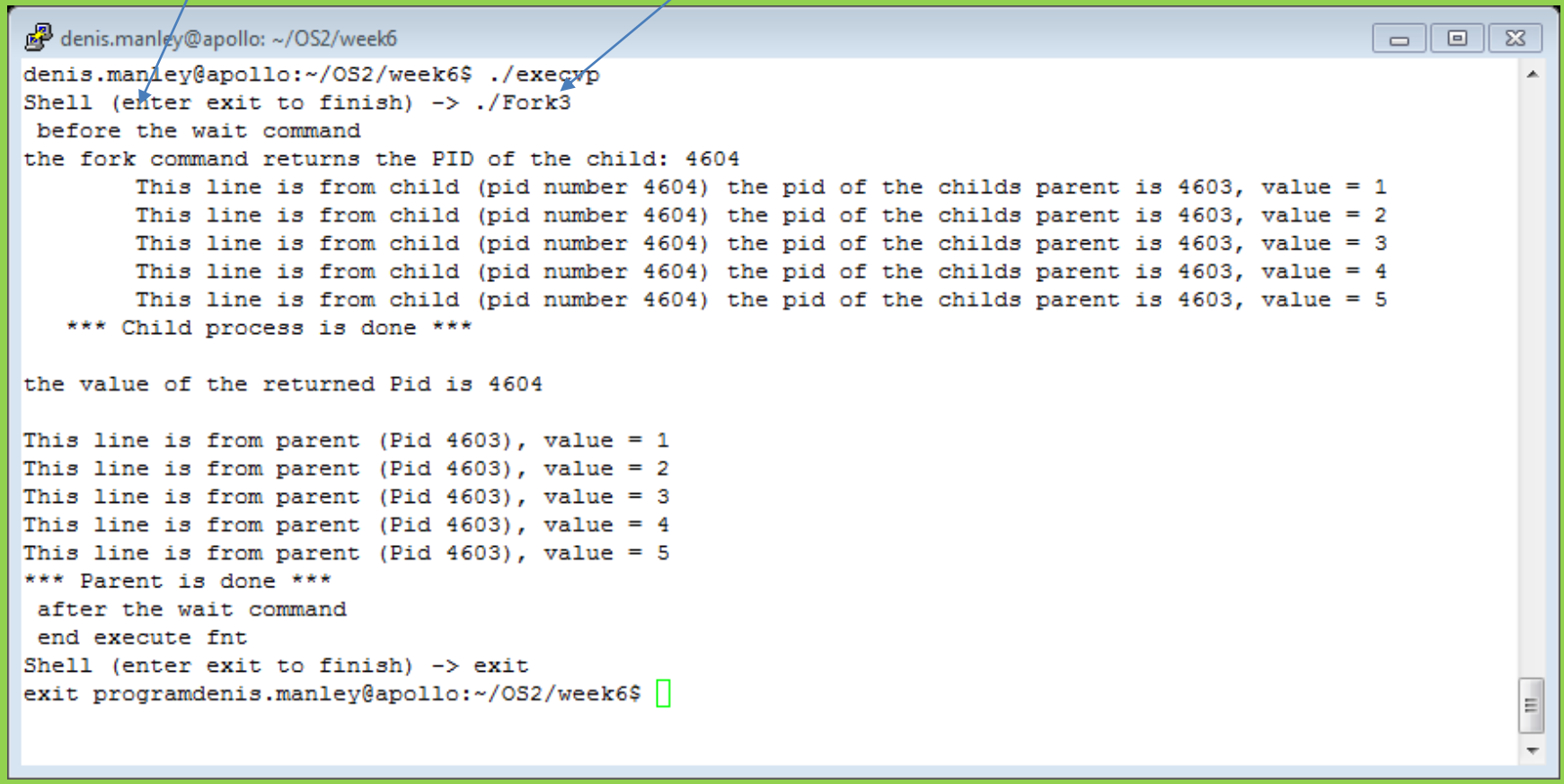
denis.manley@apollo: ~/OS2/week6

```
/* ------------------------------------------------------------------ */

void  execute(char **argv)
{
     pid_t  pid;
     int    status;

 //    printf("test execute fnt\n");

     if ((pid = fork()) < 0) {     /* fork a child process           */
          printf("*** ERROR: forking child process failed\n");
          exit(42);
     }
     else if (pid == 0) {          /* for the child process:         */
          if (execvp(*argv, argv) < 0) {     /* execute the command  */
               printf("*** ERROR: exec failed\n");
               exit(1);
          }
     }
     else {                                  /* for the parent:      */
                printf(" before the wait command \n");
          while (wait(&status) != pid);       /* wait for completion  */
          printf (" after the wait command \n");
     }

        printf(" end execute fnt\n");
}
                                                          58,1          68%
```

argv: An array of strings implemented a array of char* : argv is **

Sample output execvp.c: it uses exec() command to run the program *./Fork3*.

"Shell (enter exit to finish) is equivalent to the "linux prompt".



remove **wait** command and see what happened: explain what is happening

Function heading

- **In the execvp function we have:**
  - **int execvp(char *prog, char *argv[])**

- **These should not be confused with the**
  - **int main (int argc, char *argv[]).**

  - **The arguments in main refer to command line arguments:**
  - **argc is the number of command line arguments**
  - **Argv is an array of strings corresponing to the command line arguments: n.b. The program name is also a command line argument**

Homework Exercise

- Compile and Run the program fork1.c fork2.c fork3.c and fork4.c **using the execvp program**

- Write a program that calls the fork command twice and each child will run a different functions e.g. child one calculates the cube of a command line argument and child two calculates the factorial of the 3rd commandline argument. You will need to use the wait command to synchronise output.: refer to fork3.c wait command