Agenda

Processes

Fork/Wait

Zombies and Orphans

Process

A **process** is an instance of a running program

To run a new process, read it from disk, assign it to some memory and copy its code there

Switch to user mode and start running at the first address of the program

Starting a Program Running on System

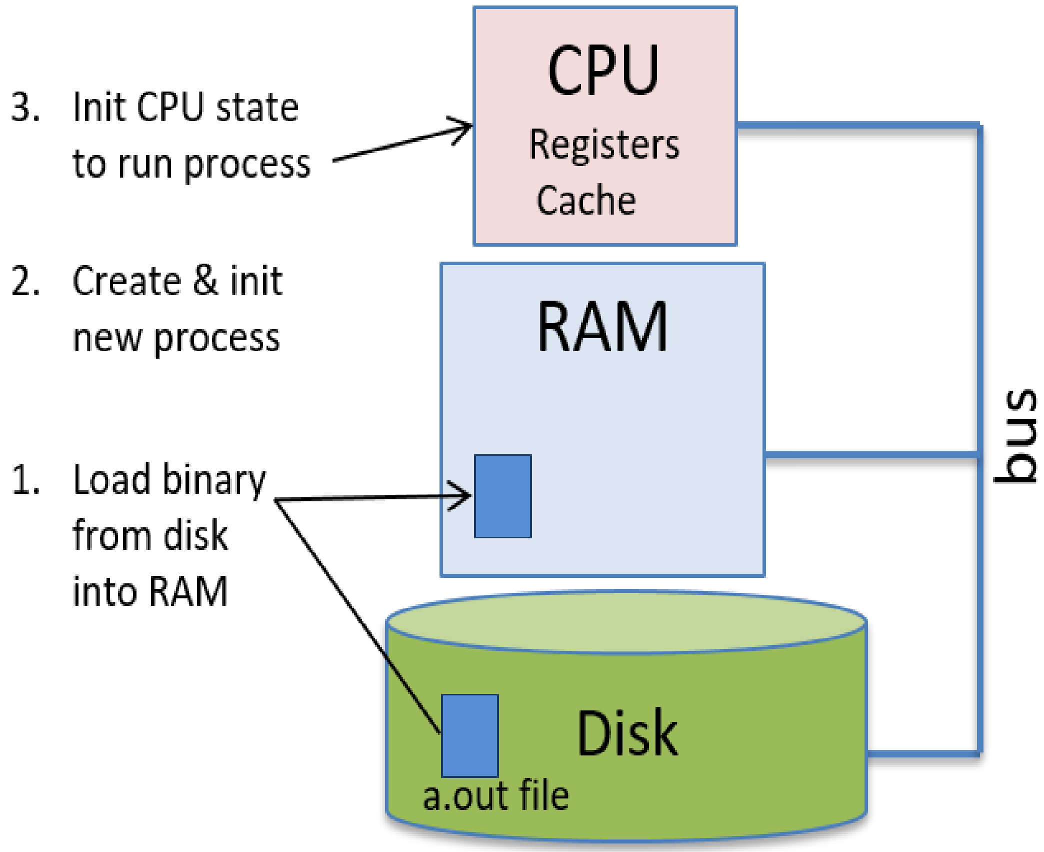


Image by Tia Newhall

Pop Quiz

What is multiprogramming?

How do processes aid with multiprogramming?

How many processes are created when we run the same program multiple times? E.g.

$ ./a.out

$ ./a.out

Process Model

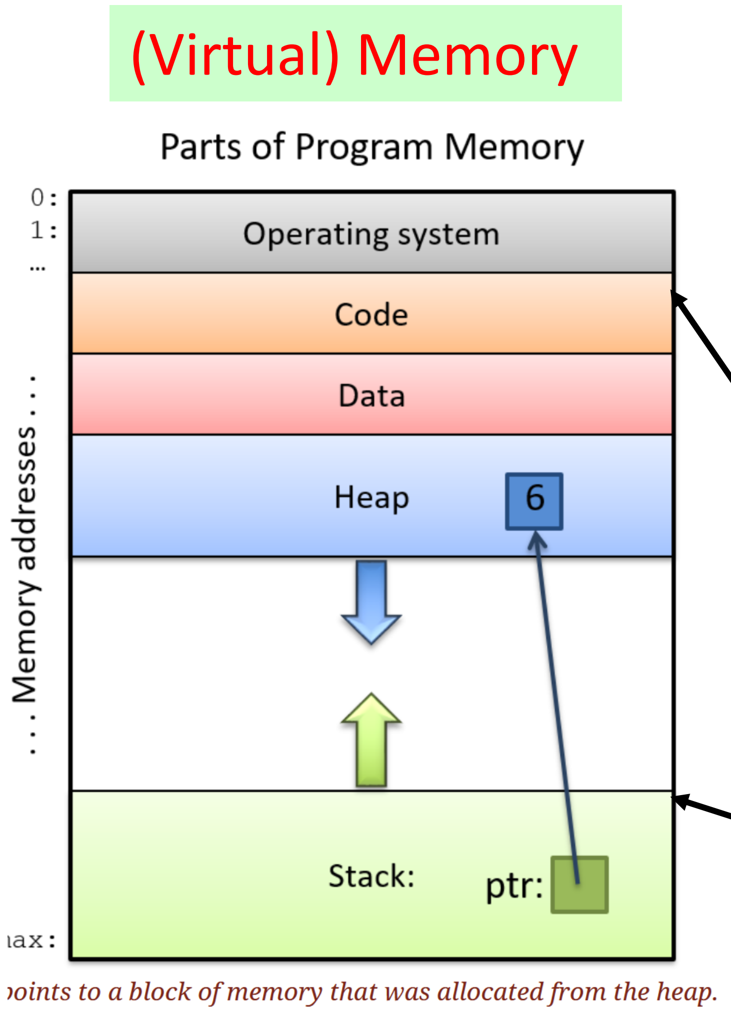
The process contains program counter (PC), registers, variables

The PC contains ______________________________

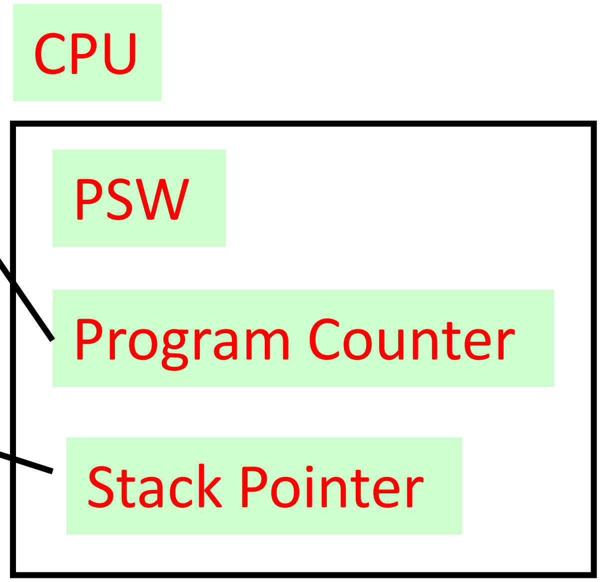
A register is ______________________________

Variables are stored where? _______________

Memory Layout of a Process

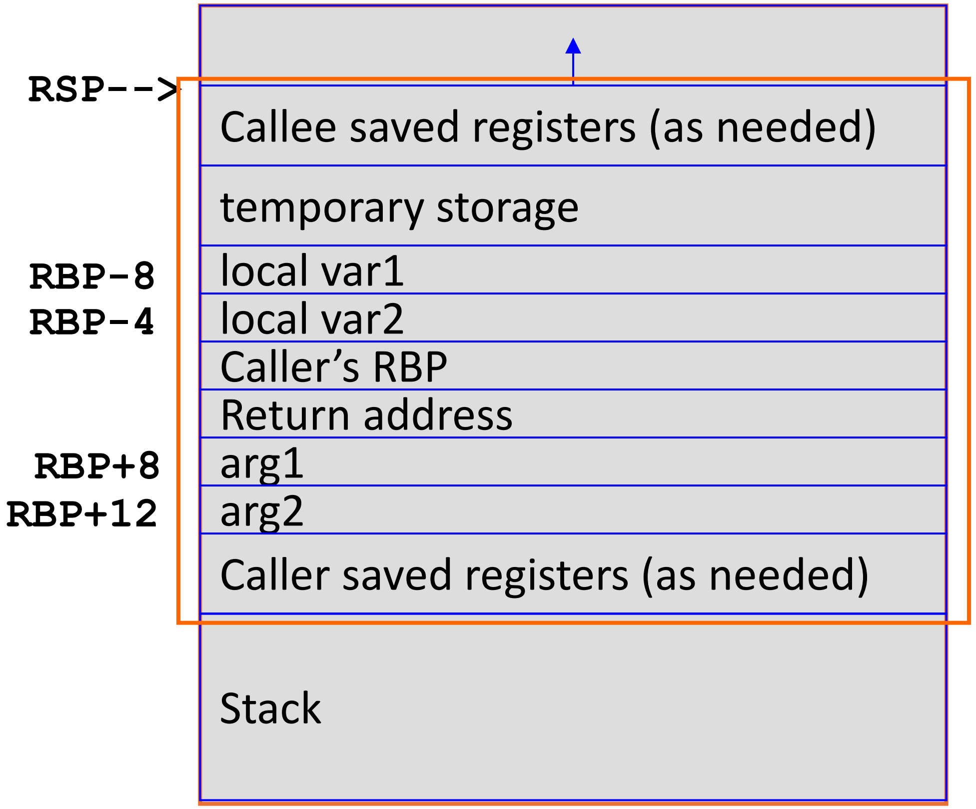


1. Where do local variables go?
2. Where do global variables go?
3. Where does the code go?
4. How is information exchanged between a program and its subroutines?



A Typical Stack Frame

- `int foo(int arg1, int arg2);`
- Two local vars



Keeping Track of Processes

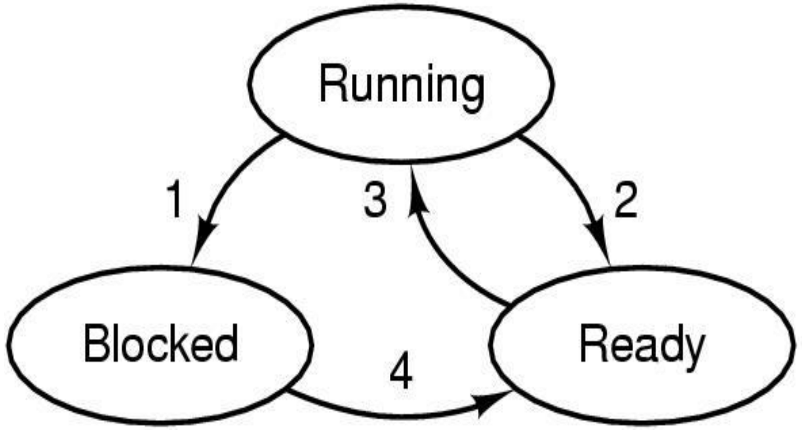
For each process, OS maintains a data structure, called the **process control block (PCB)**. The PCB provides a way of accessing all information relevant to a process:

This data is either contained directly in the PCB, or else the PCB contains pointers to other system tables.

All current processes (PCBs) are stored in a system table called the process table.

Either a linked list or an array, usually a linked list.

Process States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process states
  - Running: executing
  - Blocked: waiting for I/O
  - Ready: waiting to be scheduled

Psuedoparallism

Idea: Quickly switching between processes gives the illusion of concurrency

True parallelism: Running processes on a **multicore system**

Each process has a "lone view" of the CPU

Processes don't know whether they are running or not

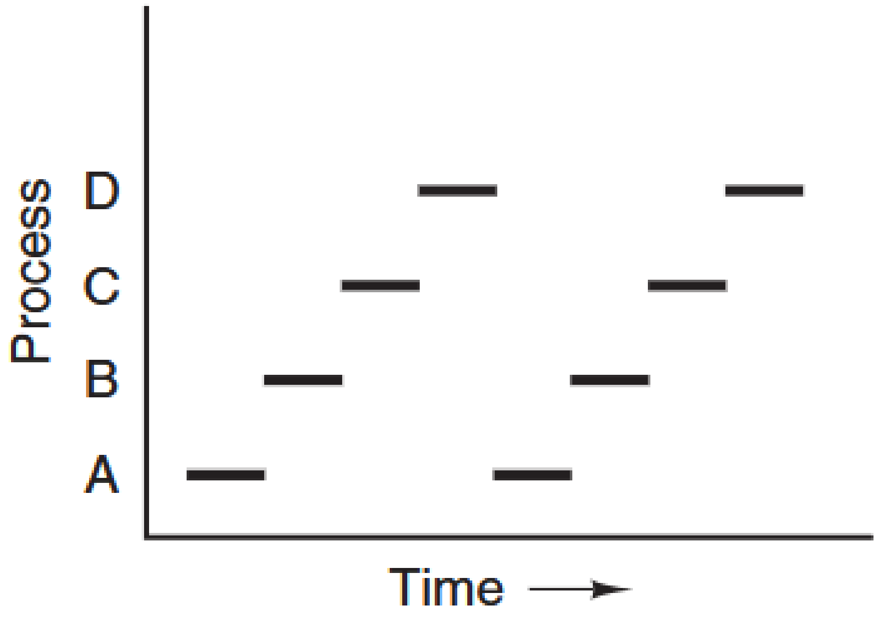
Better CPU utilization: Switch to a different process when a process is waiting (e.g. blocked)

When we switch processes, the OS must save the current process's state and then restore the state of the new process

Psuedoparallism increases CPU utilization

Example: Suppose a process is busy 20% of the time (the rest of the time, it is waiting for the user, e.g. **IO wait**)

If we have 5 of such processes, the CPU utilization is 100%



(c)

Modeling Multiprogramming

n = number of processes
p = the percentage idle

Example: if a process spends 80% of its time waiting for I/O, we need at least 10 processes to get the CPU utilization below 10%

$$\text{CPU utilization} = 1 - p^n$$

Figure 2-6 shows the CPU utilization as a function of n, which is called the **degree of multiprogramming**.

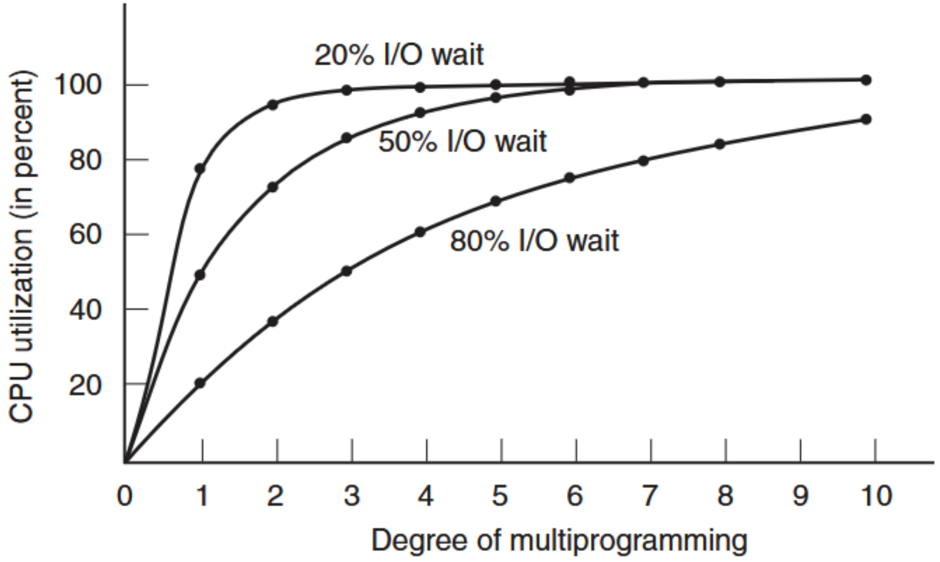


Figure 2-6. CPU utilization as a function of the number of processes in memory.

Exercise

Suppose, for example, that a computer has 8 GB of memory, with the operating system and its tables taking up 2 GB and each user program also taking up 2 GB. *How many user programs can be in memory at once on this system?*

If our processes are idle 80% of the time, what is the CPU utilization?

What if we increase the amount of memory by 8 GB?

Types of processes

System initialization (many **daemons** for processing email, printing, web pages etc.)

A **daemon** is a background process that handles requests/activity

A **foreground process** interacts with the user.

“Owns” the terminal for user input or has a GUI

A **background process** is not associated with a particular user.

List of all active processes: **`ps`** (Unix), **`Ctl-Alt-Del`** (Windows)

When are processes created?

- System initialization
- Execution of a process-creation system call by a running process.
- A user request to create a new process.
- Initiation of a batch job

When are processes terminated?

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary), due to bugs
- Killed by another process (involuntary)

When a process terminates, all the resources it owns are reclaimed by the system:

- PCB reclaimed
- its memory is deallocated
- all open files closed and Open File Table reclaimed.

Fork and wait

Reference: Process Control

UNIX provides a number of system calls for process control including:

- **`fork`** - used to create a new process
- **`exec`** - to change the program a process is executing
- **`exit`** - used by a process to terminate itself normally
- **`abort`** - used by a process to terminate itself abnormally
- **`kill`** - used by one process to kill or signal another
- **`wait`** - to wait for termination of a child process
- **`sleep`** - suspend execution for a specified time interval
- **`getpid`** - get process id
- **`getppid`** - get parent process id

Creating processes in UNIX

Processes are created in UNIX with the **`fork()`** system call.

When a Unix process is created/spawned

- a newly created process is the “child” of the “parent” process that created it
- every process has exactly one parent
- a process may create any number of child processes

Processes have a unique PID (process ID)

Index to the PCB in the process table

The `fork` System Call

- The `fork()` system call creates a "clone" of the calling process.

- Identical in every respect except
  - the parent process is returned a non-zero value (namely, the pid of the child)
  - the child process is returned zero.
  - The pid returned to the parent can be used by parent in a `wait` or `kill` system call.

- What good is this?
  - write code to behave differently if you are child

Example: Fork

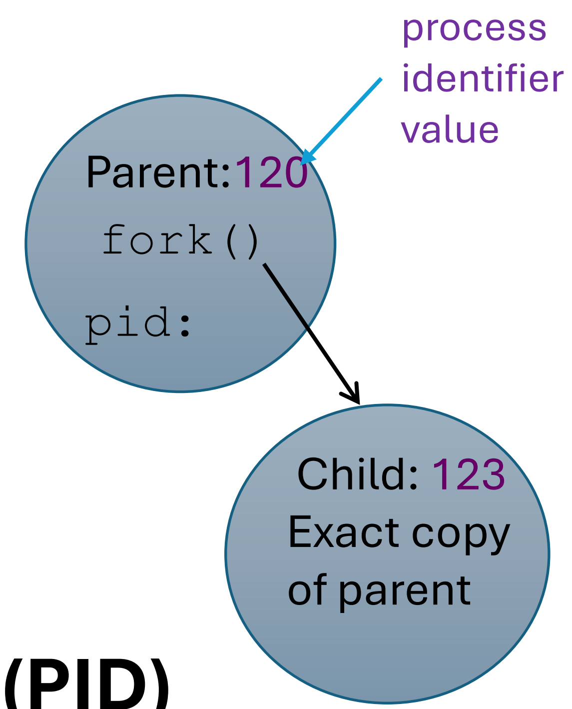
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
  pid_t ret;
  ret = fork();
  printf("ret = %d self pid = %d\n", ret, getpid());
  sleep(10);
}
```

```c
pid_t ret;
ret = fork();
```

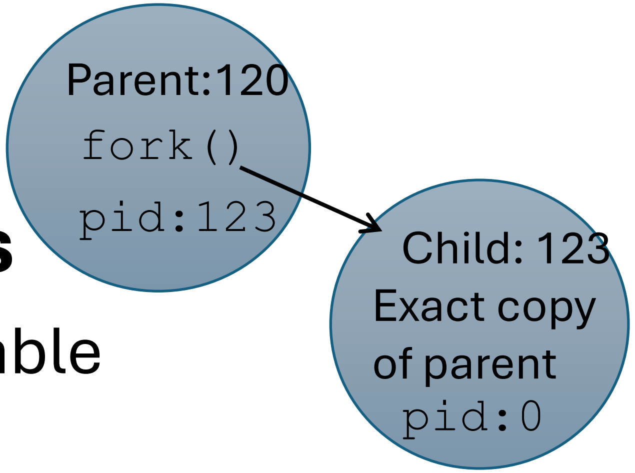
Creating a new process with fork()

Creates new process (child) that is **identical copy** of the calling process (parent):

- Analogy: An exact clone who shares your memories
- Child receives a copy of parent’s
  - address space, heap, text, data, registers, etc
  - system resources, such as open files
- But each get their own **process identifier value (PID)**



What Happens after a fork?



Parent & Child become **concurrent processes**

- Both assign return value to their copy of `pid` variable

- Who executes the `printf` statement first?
  - Depends on which gets scheduled on CPU first
  - Can vary every execution: no ordering of concurrent Pi's actions

```
ret = fork(); // both continue after call
if (ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```


22

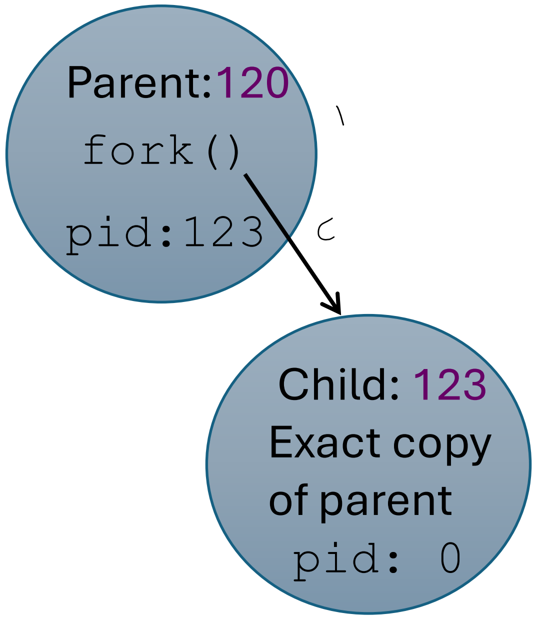
```
pid_t ret;
ret = fork();
```

Creating a new process with fork()

```
ret = fork();
// child and parent continue execution here
```

- `fork()` returns **0** to the child process
- `fork()` returns **child’s pid** to parent process

Fork is called ***once*** (by parent) but returns ***twice***
(once in parent process & once in child process)



Example

When child process is 1st scheduled to run, where is its execution point?

```
#include <unistd.h>
main(){
  pid_t pid;
  printf("Just one process so far\n");
  pid = fork();
  if (pid == 0) /* code for child */
     printf("I'm the child\n");
  else if (pid > 0) /* code for parent */
     printf("The parent, child pid =%d\n",
             pid);
  else /* error handling */
     printf("Fork returned error code\n");
}
```

Example: Visualizing concurrent processes

Both Parent and Child process can continue forking

```
void forky()
{
   printf("L0 \n");

   fork();          // parent & child cont.
   printf("L1 \n");   // both print

   fork();          // both fork new child
   printf("Bye\n");// all 4 processes print
}
```

time

Exercise

```
int main(){
    int x=0;
    fork();
    x++;
    printf("The value of x is %d\n", x);
}
```

time

Exercise

```
int main(){
    int x=0;
    int ret = fork();
   if (ret == 0) x++;
    printf("The value of x is %d\n", x);
}
```

time

Exercise

```c
int main() {
  int x=0;
  for (int i = 0; i < 2; i++) {
    int ret = fork();
    if (ret == 0) x++;
  }
  printf("The value of x is %d\n", x);
}
```

time

`wait()`

The parent process can use **wait()** to pause until a child process completes. Remove all remaining parts of exited child process from the system

`wait()` blocks until one of the caller's children terminates then returns with status or error

```
// API Calls
pid_t wait(int *child_status);
pid_t waitpid(pid_t pid, int *status);
```

Wait Example

```
void fork_and_wait() {

   int child_status;
   pid_t pid;

   if (fork() == 0) {
      printf("C\n");
      sleep(5);
   }
   else {
     printf("P\n");
     pid = wait(&child_status);
     printf(“X\n");
   }
   printf("Bye\n");
   exit(0);
}
```

```
void fork() {
   pid_t ret;
   int status;
   printf(”A\n");
   ret = fork();
   if(ret == 0) {
      printf(”B\n");
      ret = fork();
      printf(”C\n”);
      if(ret == 0) {
        printf(“D\n”);
        exit(0);
      } else {
        wait(&status);
        printf(“E\n”);
      }
   }
   else {
      wait(&status);
      printf(“F\n”);
   }
   printf(”G\n");
   exit(0);
}
```

Exercise

1. Draw Process Timeline
   1. concurrently executing dashed line
   2. no concurrent execution solid line

2. List all possible output orderings (printf ouput)

Example – child return status

```
int main() {
  int status;
  if (fork() == 0) exit(EXIT_SUCCESS); /* SIGCHLD */
  wait (&status); pexit(status);

  if (fork() == 0) abort();            /* SIGABRT */
  wait (&status); pexit(status);

  if (fork() == 0) status /= 0;        /* SIGFPE */
  wait (&status); pexit(status);
}
```

Example – child return status

```
void pexit(int status) {

  if (WIFEXITED(status)) {
    printf("Normal termination, exit status = %d\n", WEXITSTATUS(status));
  }

  if (WIFSIGNALED(status)) {
    int signal = WTERMSIG(status);
    printf("Abnormal termination: %s (%d)\n", strsignal(signal), signal);
  }

  if (WIFSTOPPED(status)) {
    printf("Stopped signal number = %d\n", WSTOPSIG(status));
  }
}
```

Replacing a Process: `exec`

- The **`exec`** system call *replaces* a process with a new program
  - it does not create a new process
  - the new program is specified by the name of the file containing the executable and arguments

- The calling process stops running as soon as it calls **`exec`** if the executable can be run

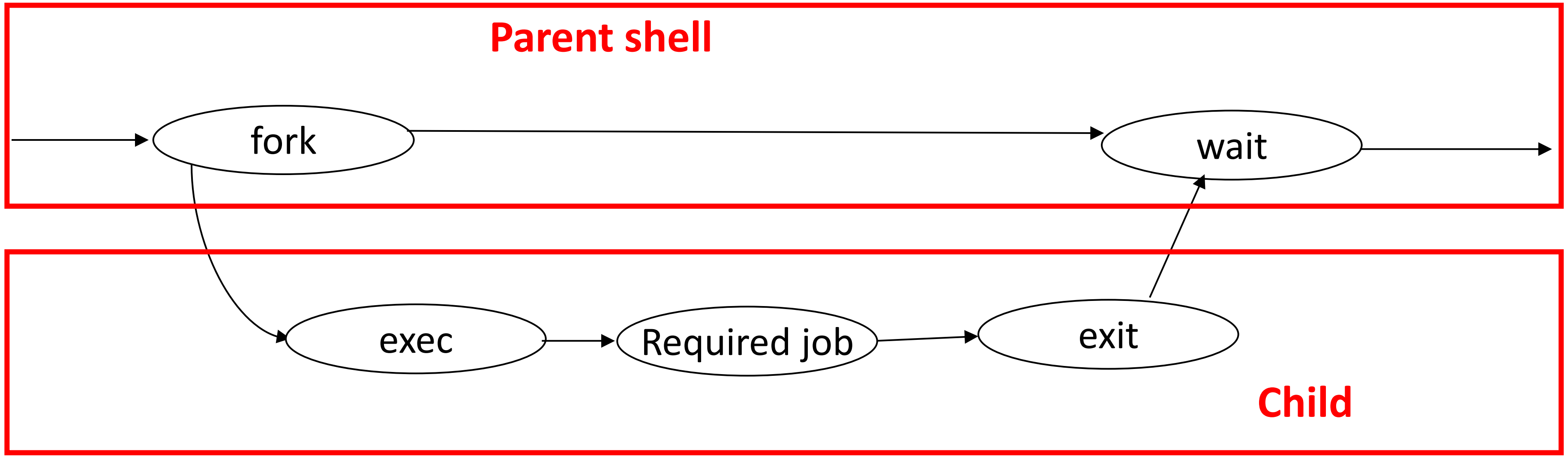
Example: spawning an application with exec

```
pid_t pid;
int status = 0;
if ( ( pid = fork() ) == 0 ) {
  /* child code: replace executable image*/
  execl("/bin/ls", “ls","-l", NULL);
}
else if (pid > 0) {
  /* parent code: wait for child to terminate*/
  wait( &status );
}
```

exec and Friends

- **exec** does not return, i.e. the program calling **exec** is gone forever!
- **exec** is really a family of system calls, each differ slightly in the way the process arguments are given
  - **execl, execlp, execle, execv, execcvp, execve**
- The l’s expect a list of pointers to strings as arguments
  - **const char *exename, const char *arg0, const char *arg1, …, const char *argn**
- The v’s expect an array of pointers to strings as arguments
- The p’s will duplicate the shell’s path searching effort
- The e’s allow an additional parameter specifying the environment variables
- All eventually make a call to **execve** (which is the real system call), the others are C lib front ends

How shell executes a command



- when you type a command, the shell **`forks`** a clone of itself
- the child process makes an **`exec`** call, which causes it to stop executing the shell and start executing your command
- the parent process, still running the shell, waits for the child to terminate

Pseudocode: Shell Program

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
    type_prompt( );                             /* display prompt on the screen */
    read_command(command, parameters);          /* read input from terminal */

    if (fork( ) != 0) {                         /* fork off child process */
        /* Parent code. */
        waitpid(−1, &status, 0);                /* wait for child to exit */
    } else {
        /* Child code. */
        execve(command, parameters, 0);         /* execute command */
    }
}
```

Figure 1-19. A stripped-down shell. Throughout this book, *TRUE* is assumed to be defined as 1.

Tannenbaum

Orphans and Zombies

An **orphan process** is a process whose parent died before it did

- gets adopted by **`init`**

A **zombie process** is a process that is waiting for its parent to accept its status code, e.g. reaps it

- The child process exited before its parent
- A parent ‘reaps’ its child when it calls \`wait\`
- Shows up with ‘Z’ in ps

Exercise: Sketch a program that creates a zombie

Use ps to see the process status

Exercise: Sketch a program that creates an orphan

Use ps to see the process status