

Module 02: Introduction to Processes and Process Management

1

What is a Process?

3.1.1 The Process

Informally, as mentioned earlier, a process is a program in execution. The status of the current activity of a process is represented by the value of the **program counter** and the contents of the processor's registers. The memory layout of a process is typically divided into multiple sections, and is shown in Figure 3.1. These sections include:

- **Text section**—the executable code
- **Data section**—global variables
- **Heap section**—memory that is dynamically allocated during program run time
- **Stack section**—temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

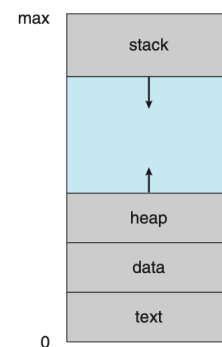


Figure 3.1 Layout of a process in memory.

2

Stack
This part of memory contains all the LOCAL VARIABLES, PARAMETERS, and BOOKKEEPING INFO ASSOCIATED WITH EACH FUNCTION CALL.

↓

↑

Heap
This part of memory contains all the DYNAMICALLY ALLOCATED MEMORY. The block grows and shrinks as needed.

Data Segment
This part of memory contains all the STATIC and GLOBAL variables associated with your process.

Text Segment
This part of memory contains all the CODE associated with your process.

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


int global_var_1;
int global_var_2;

int subroutine(int a, int b)
{
    double c;
    static int d = 10;
    if (a > d) d = a;
}

main()
{
    int a;
    int *b;

    // The POINTER VARIABLE b is in the stack.
    // The memory that b points to was "allocated" at
    // runtime by a call to malloc(). malloc() gets memory
    // in the HEAP. So, the POINTER is in the stack, and
    // memory it points to was requested from, and is in,
    // the heap.

    b = (int *)malloc(10 * sizeof(int));
}
```


 University of
CINCINNATI

3

Stack

main()
 int a int *b (variables local to main)
 book keeping info used to return control to calling function

↓

↑

Heap

Data Segment
 int global_var_1 static int d (subroutine)
 int global_var_2

Text Segment
 All the code. The program counter points to somewhere in here and it's where instructions are fetched

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


int global_var_1;
int global_var_2;

int subroutine(int a, int b)
{
    double c;
    static int d = 10;
    if (a > d) d = a;
}

main()
{
    int a;
    int *b;

    // The POINTER VARIABLE b is in the stack.
    // The memory that b points to was "allocated" at
    // runtime by a call to malloc(). malloc() gets memory
    // in the HEAP. So, the POINTER is in the stack, and
    // memory it points to was requested from, and is in,
    // the heap.

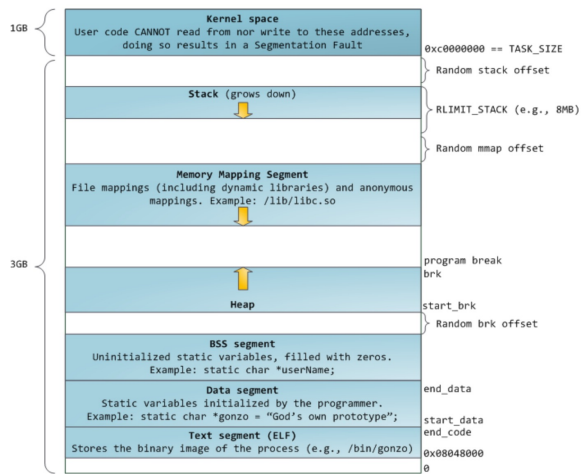
    b = (int *)malloc(10 * sizeof(int));
}
```


 University of
CINCINNATI

4

What is a Process?

A 32-bit Linux process memory map

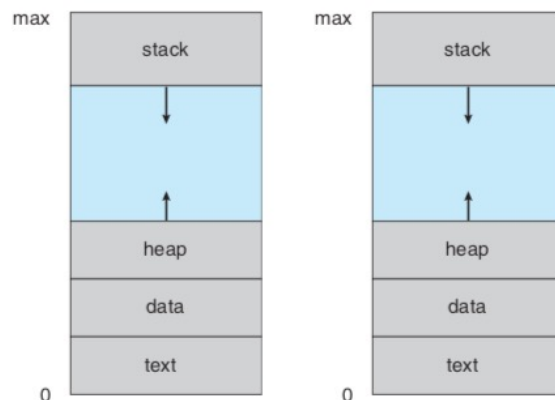


The **kernel space**, which is the memory where the CODE and DATA of the kernel is stored. In some sense... this space IS the Kernel.

The kernel space contains a **process table**, which is a data structure that contains pointers to "process control blocks" or "context blocks". The table is indexed by **Process ID (PID)**. A **PID** is a **UNIQUE** identifier for a process.

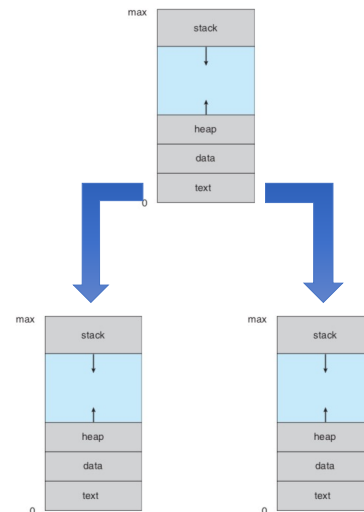
5

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs.

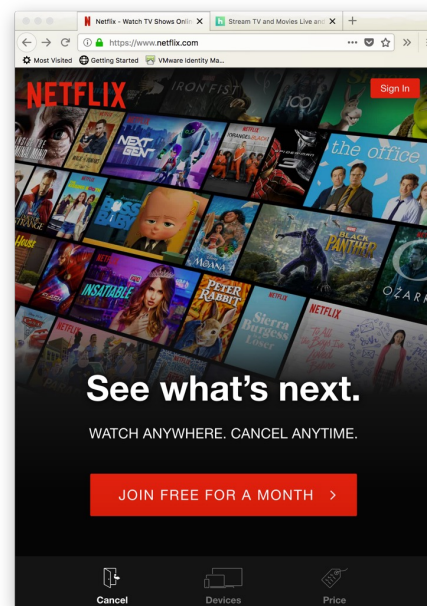


6

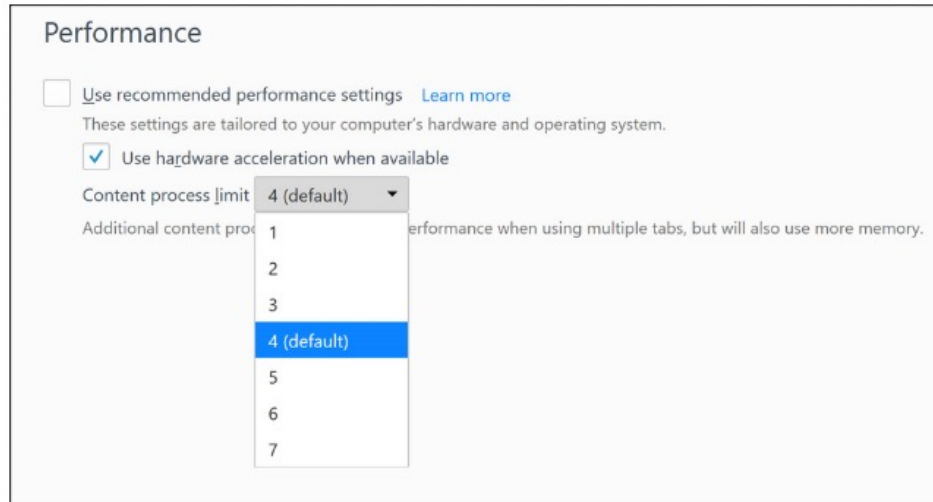
Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs.



Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs.

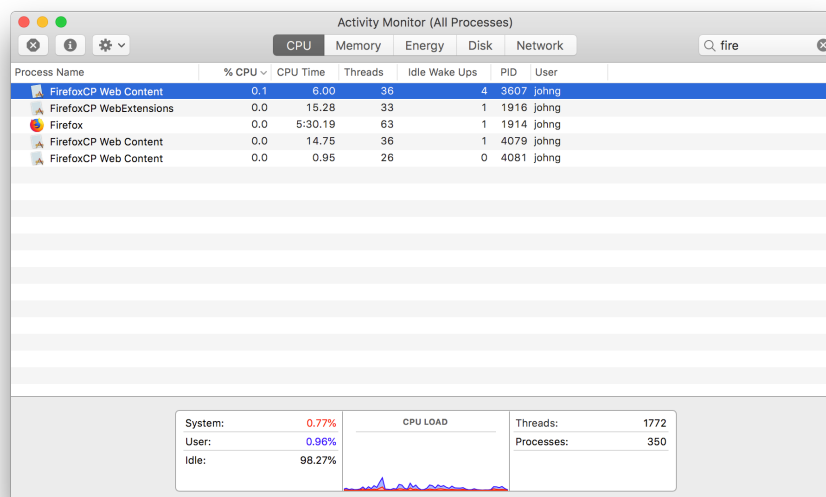


Firefox Process Count Settings



9

Firefox Process Count Settings



10

Linux Child Processes as Seen From Shell

```

johng@parallels-vm:~$ ps f
PID TTY STAT TIME COMMAND
4984 pts/1 Ss  0:00 bash
7993 pts/1 R+  0:00 \_ ps f
johng@parallels-vm:~$ xterm &
[1] 8015
johng@parallels-vm:~$ ps f
PID TTY STAT TIME COMMAND
4984 pts/1 Ss  0:00 bash
8015 pts/1 S  0:00 \_ xterm
8017 pts/17 Ss+ 0:00 | \_ bash
8078 pts/1 R+  0:00 \_ ps f
johng@parallels-vm:~$

```

11

Linux Child Processes as Seen From Shell

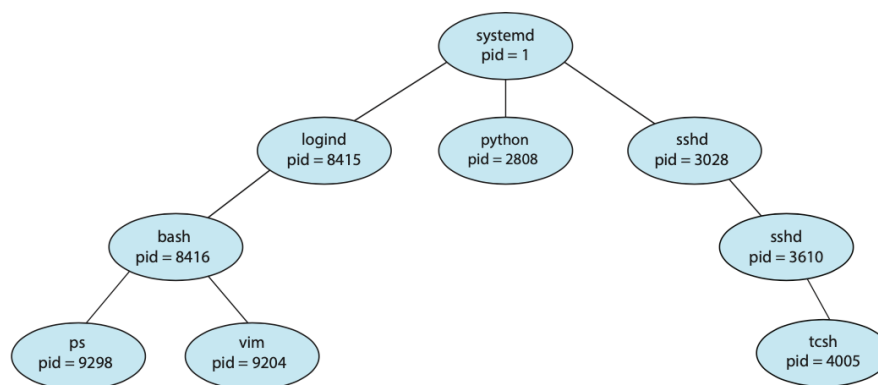


Figure 3.7 A tree of processes on a typical Linux system.

12

13

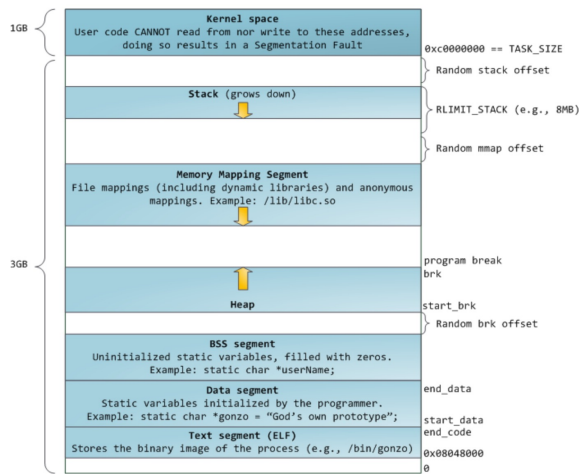
So, what's next

- We need to understand, better, what a process IS beyond its memory map
- We need to understand how a process makes more processes
- We need to understand how processes talk to one another.
- We need to understand how the OS decides what process get resources at any given time (this one will be MULTIPLE chapters in the book)

14

What's a Process?

A 32-bit Linux process memory map



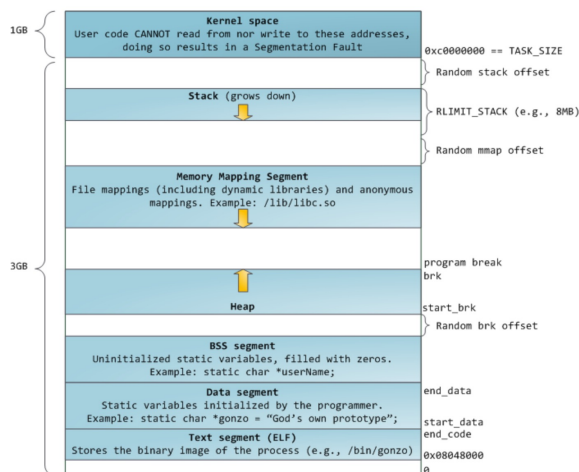
The **kernel space**, which is the memory where the CODE and DATA of the kernel is stored. In some sense... this space IS the Kernel.

The kernel space contains a **process table**, which is a data structure that contains pointers to "process control blocks" or "context blocks". The table is indexed by **Process ID (PID)**. A **PID** is a **UNIQUE** identifier for a process.

15

What's a Process?

A 32-bit Linux process memory map



The **kernel space**, which is the memory where the CODE and DATA of the kernel is stored. In some sense... this space IS the Kernel.

The kernel space contains a **process table**, which is a data structure that contains pointers to "process control blocks" or "context blocks". The table is indexed by **Process ID (PID)**. A **PID** is a **UNIQUE** identifier for a process.

A Process Control Block contains a bunch of data needed to keep track of what the process is doing, what resources its using, and other book keeping info needed by the kernel.

16

Process Control Block (PCB)

- There's lots of stuff in the PCB. For sure the PID and the PIDs of the process parent and children are in there. For now we'll focus on the process STATE, which is another item in the PCB.
- Process State is a label that defines what the "current activity" of the process is. The kernel will use process state to decide what to do with the process and when.



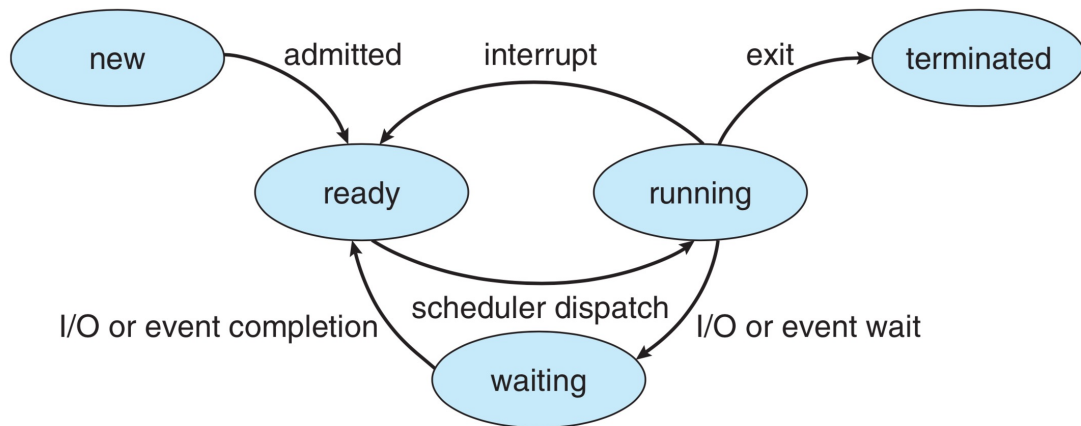
17

Process States

| | |
|--------------------|--|
| NEW: | The process has just been created, but has not yet been assigned another state. It's literally newly born. |
| RUNNING: | Instructions from the process text block are being executed. On a single processor system, only ONE processor at a time can be in this state. |
| WAITING: | The process is NOT being executed. Rather, it is waiting for some event to occur. It could be waiting on an I/O device or for a signal from the OS or another process. MANY processes can be in this state. |
| READY: | The process is not waiting or running and is waiting to be assigned to a processor so it can execute instructions (I.E. enter the RUNNING state). Many processes can be in this state. |
| TERMINATED: | The process has finished execution. It is waiting for the kernel to free the process memory, the PCB, and the processes' entry in the process table. A ZOMBIE process is a special case of a TERMINATED process. |



18

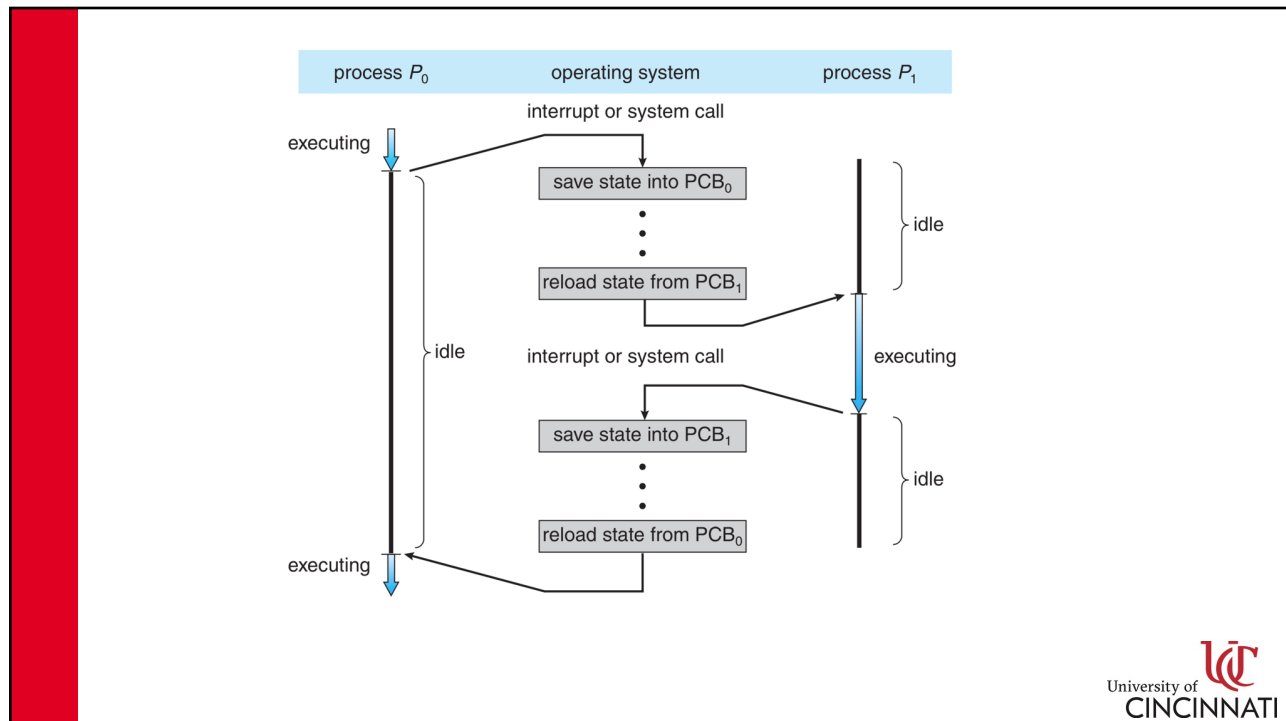


19

PCB - What Else is in There?

| | |
|---------------------------|--|
| PROCESS STATE: | The process state label |
| PROGRAM COUNTER: | The value of the program counter for THIS process. In other words, the address of the next instruction to get executed IN THE TEXT SEGMENT of the described process. This is a special case of the CPU REGISTERS . |
| CPU REGISTERS: | The values of all the CPU registers for this process. When control is given to another process, the CPU registers part of the PCB back up the register values that were there when this process was taken away from the CPU. |
| CPU SCHEDULE INFO: | Priority rating, pointers to scheduling queues, etc. (more on this later) |
| MEMORY MAP INFO: | Virtual memory information (more on this later). |
| ACCOUNTING INFO: | A record of resources already used (CPU time, real time, user account numbers, children and parent PIDs, etc.) |
| I/O INFO: | I/O Devices currently being used by the process including locators for open files |

20



21

Process Scheduling

The objective of **multiprogramming** is to have some process running at all times. Otherwise, the CPU is not being used fully.

The objective of **time sharing** is to switch the CPU (or CPUs) among processes so frequently that users have the illusion that all of the processes react to them in a timely manner.

The objective of the **process scheduler** is to choose which process is assigned to a CPU at what time. A high quality process scheduler will maintain full, or near full, CPU utilization and maintain the illusion of immediate response from processes. It will also ensure that no single process goes forever without CPU access (starvation) and that processes don't get into unwinnable resource wars (deadlocks).

22

Process Scheduling

The kernel usually moves PCB references from scheduling queue to scheduling queue.

You can think of a schedule queue as a “line” of processes that are waiting for access to a resource (CPU, I/O device, signal from another process, etc.). The kernel moves PCB references among the queues according to process need and access rules enforced by the kernel.

Different OSes have different numbers of and types of scheduling queues. Some common ones are:

- Job queue: ALL the processes in the system
- Ready queue: The processes that are immediately ready to execute
- Device queue: Processes waiting for access to an I/O device. There is one queue per device.



23

A Sidenote on Threads

A **simple process** has a SINGLE line of execution inside of each process. There is a SINGLE program counter and the CPU executes instructions pulled from the text segment one at a time. The instructions access memory in the data, heap, or stack segments.

A **threaded process** has MULTIPLE lines of execution inside of each process. There are MULTIPLE program counters executing instructions from different addresses inside the text segment. ALL lines of execution share the same data, heap, and stack segments inside the process record. Generally this is most useful on multi-core systems in which a different computational core can be assigned to each thread. **YES, THIS IS COMPLICATED AS THREADS CAN CLOBBER ONE ANOTHER.** We'll talk about the how and why of threads later. Sometimes threads are called “lightweight processes” because they have multiple lines of execution, but are “lighter” to context switch.



24

Context Switching

- A context switch is the process by which CPU access is switched from one process to another
- A context switch involves saving the state (a state store) of one process and a state restore of another (a state restore). In this case, "state" means the values of all the CPU registers along with other active information associated with a process that gets "written over" when processes switch.
- Context switching happens frequently on a multi-programmed system. It is imperative that the process be efficient.



25

Creation of Processes

Processes can in most cases execute concurrently via time-sharing. Also, they are created and destroyed dynamically.

- processes created by user request via a shell or GUI
- processes created by the system to help with specific jobs (system processes)
- processes created by other processes to modularize functionality

We generally conceive as all the processes existing in a process tree. The OS spawns the very first "mother of all processes". On Unix this process is called init (old systems) or systemd (new systems).



26

Communication Among Processes

Processes can communicate among themselves

- Shared Memory Models
- Messaging Models
- Interprocess Signaling
- Pipes / Files

27

Linux Child Processes as Seen From Shell

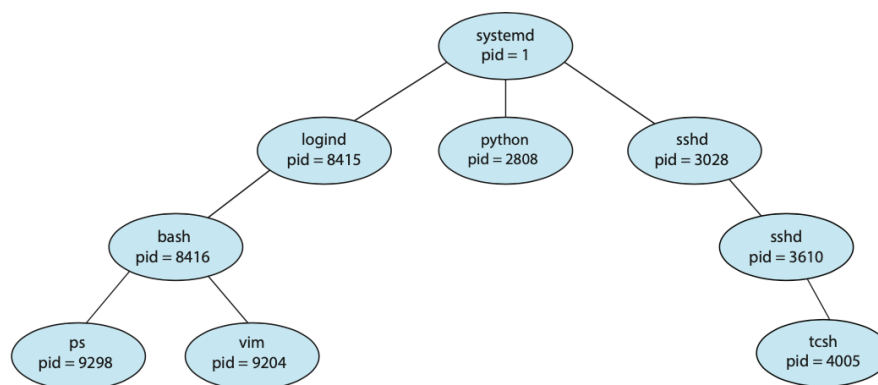


Figure 3.7 A tree of processes on a typical Linux system.

28

Process Creation

To build out a process tree, one needs kernel support to CREATE, EXECUTE, and DESTROY processes. Further, the kernel must keep track of child and parent relationships in the tree to facilitate interprocess communication and process management.

When one process creates another, there are two execution options

1. The parent continues to execute concurrently with the children
2. The parent waits until some or all of its children have terminated

When one process creates another, there are two memory space options

1. The child is (at least initially) a clone of the the parent. It has copies of the same data and text as the parent.
2. The child process has a new program loaded into it and “fresh” data



29

Process Creation

How are new processes created and controlled? It depends. It generally involves a set of system calls...

- In Unix-like OSES, one uses the **fork()** system call.
- **fork()** makes a CLONE of the current process. The CLONE starts with the copies of the text and data segments of the original program. For the original (parent) process, **fork()** returns the PID of the CHILD process. For the child process, **fork()** returns a PID of zero (that's how it knows it's the clone and not the original)
- Initially, both the parent and the child (clone) are in the “ready” queue and could be scheduled for CPU at any point
- The child (or the parent) COULD at any time load new code into its text segment with the **exec()** system call.
- The parent COULD at any time put itself in the WAIT queue and sleep until one or more children complete their work using the **wait()** system call.
- Any process can end itself by using the **exit()** or **_exit()** system calls as appropriate.
- Processes can control one another (if allowed) with Interprocess Signals. They can also share information via models previously mentioned that will be developed in detail later.



30

Process Creation

So how does this work in practice?

- 1) A process makes a clone of itself. The clone contains all the data and code of the original. The clones, however, CAN TELL that they are clones, so they can choose to act differently (using the same code) than the parent.
- 2) Because the clone starts with the data of the original, it can access any context that it SHOULD inherit from the parent. For example if the parent were a shell, and you forked a copy of the shell that was supposed to run another program, it's good that the shell's environment went to the copy so that the program that is launched by the clone has all the shell environment info.
- 3) The clone could do the same thing as the parent in parallel, OR it **could load NEW code into its text segment**, becoming something else but still maintaining access to inherited information
- 4) The parent could run in parallel or it could go to sleep while all the children are running.

31

An Example (from Wikipedia)

```
int main(void)
{
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        printf("Hello from the child process!\n");
        _exit(EXIT_SUCCESS);
    }
    else {
        int status;
        (void)waitpid(pid, &status, 0);
    }
    return EXIT_SUCCESS;
}
```

32

An Example (from Wikipedia)

```
int main(void)
{
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        printf("Hello from the child process!\n");
        _exit(EXIT_SUCCESS);
    }
    else {
        int status;
        (void)waitpid(pid, &status, 0);
    }
    return EXIT_SUCCESS;
}
```

The first statement in main calls the `fork()` system call to split execution into two processes. The return value of `fork` is recorded in a variable of type `pid_t`, which is the POSIX type for process identifiers (PIDs).



33

An Example (from Wikipedia)

```
int main(void)
{
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        printf("Hello from the child process!\n");
        _exit(EXIT_SUCCESS);
    }
    else {
        int status;
        (void)waitpid(pid, &status, 0);
    }
    return EXIT_SUCCESS;
}
```

If `fork()` returns a negative number, it means that it was impossible to create a child. In this case, the “original” process should terminate itself.



34

An Example (from Wikipedia)

```
int main(void)
{
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        printf("Hello from the child process!\n");
        _exit(EXIT_SUCCESS);
    }
    else {
        int status;
        (void)waitpid(pid, &status, 0);
    }
    return EXIT_SUCCESS;
}
```

If you even got this far, then `fork()` did not return an error code. This means you are one of two processes... the original, or the clone. If your PID is zero, you are the clone. If your PID is non-zero, you are the parent and the PID is the PID of the child you just created. In this case, if you're the child, you just say "hello" and terminate with `_exit()`.



35

An Example (from Wikipedia)

```
int main(void)
{
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        printf("Hello from the child process!\n");
        _exit(EXIT_SUCCESS);
    }
    else {
        int status;
        (void)waitpid(pid, &status, 0);
    }
    return EXIT_SUCCESS;
}
```

If you got here, you didn't run the child code because... you're not the child. Instead you run the parent code. In this case, the parent code says "I'm gonna put myself on the wait queue and stay out of the way until I get a signal from the child I just created (whose PID is in the variable `pid`). When that child is done, I'll "wake up" and go the ready queue. Eventually I'll execute and shut myself down too.



36

37

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

int main()
{ pid_t who_am_i;
  FILE *fp;
  char c = '@';

  fp = fopen("fork.c", "r");

  who_am_i = fork();

  if (who_am_i == 0)
  { while (!feof(fp))
    { c = getc(fp);
      putchar(c);
    }
  }
  else
  { printf("Message from the parent process. c == %c\n",c);
  }
}
```

38

```

johng@zuul test % gcc fork.c
johng@zuul test % ./a.out
Message from the parent process. c == @
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{ pid_t who_am_i;
  FILE *fp;
  char c = '@';

  fp = fopen("fork.c", "r");

  who_am_i = fork();

  if (who_am_i == 0)
  { while (!feof(fp))
    { c = getc(fp);
      putchar(c);
    }
  }
  else
  { printf("Message from the parent process. c == %c\n",c);
  }
}
johng@zuul test %

```

39

Process Creation



Yeah.... This is how it actually works.... Except that generally all the text segments are not Smith...

40

Communication Among Processes

Processes can communicate among themselves

- Shared Memory Models
- Messaging Models

We'll look at these conceptually

- Interprocess Signaling
- Pipes / Sockets / Files

We'll do a little coding with these

An Inter-Process Communication (IPC) method is any method that allows data to be transferred from one process to another. There are MANY kinds of IPC. We'll consider a few, and those in the Linux/Unix style.

41

Communication Among Processes

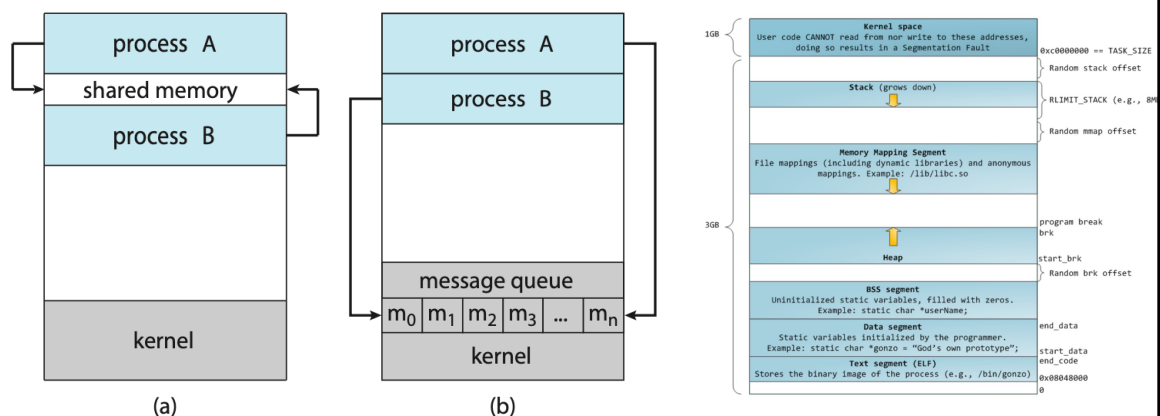


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

42

Producer Consumer Model

```

item next_produced;

while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}

```

Figure 3.12 The producer process using shared memory.

```

item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_consumed */
}

```

Figure 3.13 The consumer process using shared memory.

in is a SHARED variable in both processes
out is a SHARED variable in both processes
buffer is a SHARED variable pointer that points to a block of memory also shared by both processes. The buffer has **BUFFER_SIZE** slots in it

43

Communication Among Processes

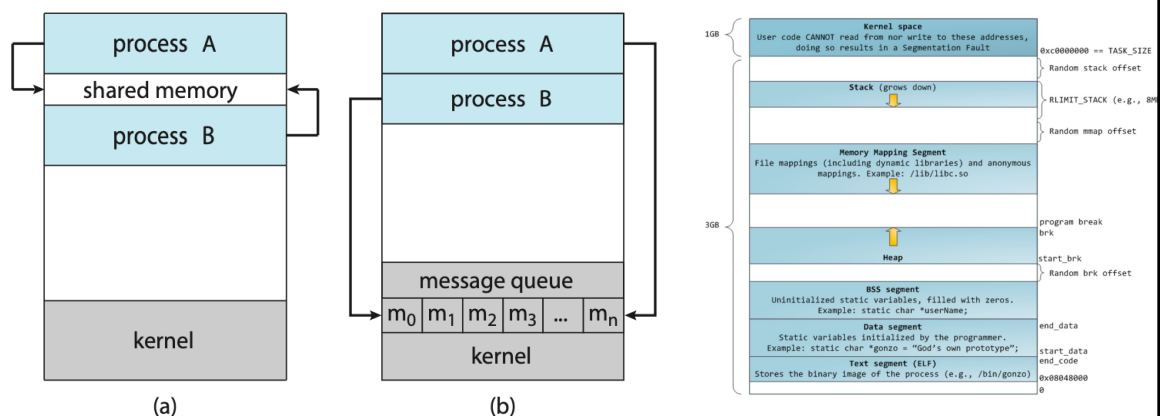


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

44

IPC In Message-Passing Systems

We're going to rely *heavily* on your reading section 3.6 – as most of the information here is definitional / conceptual

In short:

- Message passing systems are mediated by the kernel, which will handle ALL the difficult synchronization for you. Hurray!
- Message passing systems add much overhead when moving LARGE blocks of data from process to process. BOO!
- Direct vs. Indirect
 - Mailbox vs. process-to-process
- Synchronous vs. Asynchronous
 - Blocking vs. non-blocking receives and sends
- Automatic vs. Explicit Buffering
 - zero, specified, or unbounded



45

IPC In Shared Memory Systems

We're going to rely on reading AND on future work with shared memory in the context of threads. Much of what we'll care about there we'd care about for shared memory in processes, so we'll let that discussion hang a bit...

In short:

- Shared memory is DANGEROUS because there is NO protection for concurrent access. When the order of operations is non-deterministic, you're ASKING for data corruption unless you use specific tools and techniques to prevent it. More on that when we do threads.
- Shared memory is GREAT when you want to share large memory structures as no explicit kernel intervention is required. Threads take this even one step further and get rid of the need for context switches (we'll return to this). The price you pay is that YOU have to be responsible for data consistency.



46