Processes

Processes

Picture this: You started on assignment 8, happily coding away. You don't have a worry in the world, you have all the resources of lectura available to you. Is this really the case?

No! There are many *processes* running at the same time taking up different resources!

We are working in a unix operating system, what is the purpose of thisOS?

- *Resource management*: coordinate resource usage by (possibly competing) applications; isolation and communication.
- *Abstractions*: provide advanced (or more) facilities that everyone needs and the hardware doesn't provide.

The OS allows several users to be working at the same time, as if each had a private personal machine. Or, one user can be doing many things at the same time. To keep track of everything, notion of *process* was invented. It’s hard to keep processes from interacting in bad ways.

Processes help with resource management by allowing us to split up resources between different applications running each as its own process.

Processes

A process is an occurrence of an executing program that contains information about the programs state.
A process is not the same thing as an application, such as ls or gcc, it is both more and less than this concept.

- It's less because a process could only be part of the state, for example applications such as gcc derive many children processes from the main one.
- It's more because an application is only part of the state -- what it does depends on the execution, many processes can be derived from the "ls" command. I.e. something different happens depending on where it is used.

When your operating system starts on a linux (Note: not unix) machine, there is a process called *init.d* that gets created. That process is a special one handling signals, interrupts, and a persistence module for certain kernel elements. Whenever you want to make a new process, you call *fork* (to be discussed later) and use another function to load another program.

Processes

Processes are very powerful but they are isolated!

That means that by default, no process can communicate with another process.

This is very important because if you have a large system (like lectura) then you want some processes to have higher privileges (monitoring, admin) than your average user.

One certainly doesn't want the average user to be able to bring down the entire system either on purpose or accidentally by modifying a process.

A process is always created and then given the permissions it needs.

Processes

If I run the following code,

```
int secrets = 0; // maybe defined in the kernel or else where

secrets++;

printf("%d\n", secrets);
```

On two different terminals (Note: every terminal is its own process), what would this print out?

As you would (hopefully) guess they would both print out 1 not 2.

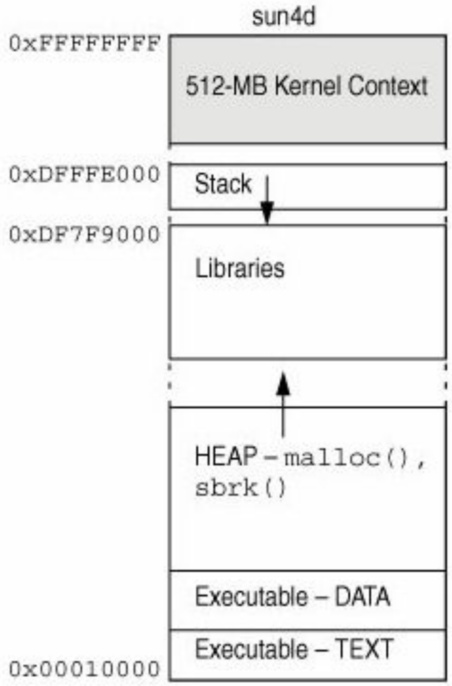
Point: even though we are running the same code, there is different state between the two processes.

Even if we changed the code to do something really hacky (apart from reading the memory directly) there would be no way to change another process' state.

Processes

When a process starts, it gets its own address space. Meaning that each process gets (for Memory)

- A stack
- A heap
- A data segment
- A text segment



Processes

To keep track of all these processes, your operating system gives each process a number and that process is called the PID, process ID.

Processes could also contain

- Mappings
- State
- File Descriptors
- Permissions

On a UNIX system, we can view all of our processes running with the ps command. From the man page:

```
ps - report a snapshot of the current processes.
```

Processes

When I run **ps**, I see the following

```
% ps
  PID TTY          TIME CMD
  488 pts/17   00:00:00 bash
11474 pts/17   00:00:00 ps
```

This reports several things.

PID - the id of the process
TTY - the terminal that executed the process
TIME - total CPU usage, time it has run
CMD - what command it is

Processes

We can use the command **ps -aux** to list all processes we are running (or -f for full).

```
% ps -aux | wc
    317    3983   58314
% ps -aux |  head -2
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  36208  8964 ?        Ss   Jun19   0:06 /sbin/init
% ps -aux | grep dpdicken | head -2
dpdicken   488  0.0  0.0  25384  4012 pts/17   Ss+  Jul31   0:07 bash
dpdicken  1616  0.0  0.0 742780 26760 ?        Sl   Jul07   0:04
/usr/lib/x86_64-linux-gnu/unity-scope-home/unity-scope-home
% ps 1616
  PID TTY      STAT   TIME COMMAND
 1616 ?        Sl     0:04
/usr/lib/x86_64-linux-gnu/unity-scope-home/unity-scope-home
```

Note how using -aux gives us a more verbose report on the processes.

Processes

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command.

You can also end a process by getting its PID and using the kill command.

```
% ps -aux | grep dpdicken | head -2
dpdicken   488  0.0  0.0  25384  4012 pts/17   Ss+  Jul31   0:07 bash
dpdicken  1616  0.0  0.0 742780 26760 ?        Sl   Jul07   0:04
/usr/lib/x86_64-linux-gnu/unity-scope-home/unity-scope-home
% kill 488
%
```

The process with pid 488 has now been killed.

```
% kill 1
bash: kill: (1) - Operation not permitted // What happened?
```

Processes

The command kill sends the specified signal (remember, IPC?) to the specified process or process group.

If no signal is specified, the TERM signal is sent. The TERM signal will kill processes which do not catch this signal.

For other processes, it may be necessary to use the KILL (9) signal, since this signal cannot be caught.

We can send the signal KILL(9) by using the -9 option in kill like so:

```
% kill -9 488 // Note how it came before the PID
%
```

Processes

Processes can exist in two environments -- background or foreground.

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen. This happens with any command we run from the terminal.

You may have had this issue before: You run a program in the foreground that takes awhile to finish. You are not able to run any other commands while it completes because the prompt is not available until it finishes.

This problem is resolved with the idea of background processes.

Processes

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (**&**) at the end of the command.

```
% ls *.c & // it is running in the background

[1] 122123
test.c

[1]+  Done                    ls *.c
%
```

Here, if the **ls** command wants any input (which it does not), it goes into a stop state until we move it into the foreground and give it the data from the keyboard.

Processes

If you are running a command in the foreground, you can hit ‘CTRL-Z’ to suspened it (not kill stopped).

Once it is no longer in the foreground, you can type bg to send it to the background.

You can then use the **jobs** command to list all processes in the background.

```
% ls * -R * > testingfas
^Z
[1]+  Stopped                 ls * -R * > testingfas
% bg
[1]+ ls * -R * > testingfas &
% jobs
[1]+  Running                 ls * -R * > testingfas &
```

Processes

We can then use the foreground command (**fg**) to bring a job back.

```
% jobs
[1]+  Running                 ls * -R * > testingfas &
[2]+  Running                 ls * -R ~/* > testingfas2 &
% fg %2
% jobs
[1]+  Running                 ls * -R * > testingfas &
% fg
% jobs
%
```

Processes

Each unix process has two ID numbers assigned to it: The Process ID (pid) and the Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell they were run in as their parent.

Normally, when a child process is killed, the parent process is updated via a **SIGCHLD** signal. Then the parent can do some other task or restart a new child as needed.

However, sometimes the parent process is killed before its child is killed. In this case, the "parent of all processes," the **init** process, becomes the new PPID (parent process ID). In some cases, these processes are called orphan processes.

When a process is killed, a **ps** listing may still show the process with a **Z** state. This is a “zombie” or defunct process. The process is dead and not being used.

These processes are different from the orphan processes. They have completed execution but still find an entry in the process table and are waiting to be cleaned up.

Processes

Daemons are system-related background processes that often run with the permissions of root and service requests from other processes.

To be precise, a daemon is a process that runs in the background, usually waiting for something to happen that it is capable of working with. Examples:

A printer daemon waiting for print commands.

On lectura, fingerd, provides an interface for the finger command.

If you have a program that calls for lengthy processing, then it’s worth to make it a daemon and run it in the background.

Processes

The **top** command is a very useful tool for quickly showing processes sorted by various criteria.

It is an interactive diagnostic tool that updates frequently and shows information about physical and virtual memory, CPU usage, load averages, and your busy processes.

Let's take a look on lectura.

Processes

A final note:

Background and suspended processes are usually manipulated via **job number (job ID)**. This number is different from the process ID and is used because it is shorter.

In addition, a job can consist of multiple processes running in a series or at the same time, in parallel. Using the job ID is easier than tracking individual processes.

Processes - In C programs

We are able to create our own processes in a C program using fork(). Man page:

```
NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork() creates a new process by duplicating the calling process.  The new
process, referred to as the child, is an exact duplicate of the calling process,
referred to as the parent, except for the following points:
```

Processes - In C programs

The fork system call clones the current process to create a new process. It creates a new process (the child process) by duplicating the state of the existing process with a few minor differences (discussed later).

The child process does not start from main. Instead it returns from fork() just as the parent process does.

Note: Process forking is a very powerful (and very dangerous) tool. If you mess up and cause a fork bomb (explained later on this page), you can bring down the entire system. To reduce the chances of this, limit your maximum number of processes to a small number e.g 40 by typing ulimit -u 40 into a command line.

Processes - In C programs

Here is a simple example:

```
int main() {
  printf("I'm printed once!\n");
  fork();
  // Now there are two processes running
  // and each process will print out the next line.
  printf("You see this line twice!\n");
}
```

Execution:

```
% ./a.out
I’m printed once!
You see this line twice!
You see this line twice!
```

Processes - In C programs

The following program prints out 42 twice - but the fork() is after the printf!? Why?

```
#include <unistd.h> /*fork declared here*/
#include <stdio.h>
int main() {
   int answer = 84 >> 1;
   printf("Answer: %d", answer);
   fork();
   return 0;
}
```

Processes - In C programs

Okay, so our created processes are starting at the same place in the code. How do we make processes that execute different code?

Check the return value of fork(). Return value -1 is fork failed, 0 is in child process, positive is in parent process (and the return value is the child process id).

Here's one way to remember which is which:

The child process can find its parent - the original process that was duplicated - by calling getppid() - so does not need any additional return information from fork(). The parent process however can only find out the id of the new child process from the return value of fork.

Processes - In C programs

Here is an example that detects which process it is:

```
pid_t id = fork();
if (id == -1) exit(1); // fork failed
if (id > 0) {
// I'm the original parent and
// I just created a child process with id 'id'
// Use waitpid(id) to wait for the child to finish
} else { // returned zero
// I must be the newly made child process
}
```

Processes - Sidenote: fork bomb

A 'fork bomb' is when you attempt to create an infinite number of processes. A simple example is shown below:

```
while (1) { fork(); }
```

This will often bring a system to a near-standstill as it attempts to allocate CPU time and memory to a very large number of processes that are ready to run.

Comment: System administrators don't like fork-bombs and may set upper limits on the number of processes each user can have or may revoke login rights because it creates a disturbance in the force for other users' programs.

You can also limit the number of child processes created by using setrlimit().

fork bombs are not necessarily malicious - they occasionally occur due to student coding errors.

Processes - In C programs

Okay, how do we wait for a child process to finish?
Use waitpid() (or wait()).

```
NAME
       wait, waitpid, waitid - wait for process to change state
SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>
       pid_t wait(int *status);
       pid_t waitpid(pid_t pid, int *status, int options);
       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

DESCRIPTION
     All of these system calls are used to wait for state changes in a child of
     the calling process, and obtain information about the child whose state has
     changed.
```

Processes - In C programs

Here is an example using waitpid:

```
int main() {
    pid_t child_id = fork();
    if (child_id == -1) { perror("fork"); exit(EXIT_FAILURE);}
    if (child_id > 0) {
      // We have a child! Get their exit code
      int status;
      waitpid( child_id, &status, 0 );
      // code not shown to get exit status from child
    } else { // In child ...
      // start calculation
      exit(123);
    }
```

Processes - In C programs

The concept of checking the return value of fork and branching based on it is fairly cumbersome.

We can actually make our child process execute different programs.

The **exec** set of functions replaces the process image with the the process image of what is being called. This means that any lines of code after the exec call are replaced. Any other work you want the child process to do should be done before the exec call.

Let's look at the man page!

The Wikipedia article does a great job helping you make sense of the names of the exec family (search exec (system call)).

Processes - In C programs

The naming scheme of the functions can be shortened like this:

The base of each is exec (execute), followed by one or more letters:

e – An array of pointers to environment variables is explicitly passed to the new process image.

l – Command-line arguments are passed individually (a list) to the function.

p – Uses the PATH environment variable to find the file named in the file argument to be executed.

v – Command-line arguments are passed to the function as an array (vector) of pointers.

Processes - In C programs

Example:

```
int main(int argc, char **argv) {
  pid_t child = fork();
  if (child == -1) return EXIT_FAILURE;
  if (child) { /* I have a child! */
    int status;
    waitpid(child , &status ,0);
    return EXIT_SUCCESS;
  } else { /* I am the child */
    execl("/bin/ls", "ls","-alh", (char *) NULL);
    perror("exec failed!");
  }
}
```

Processes - In C programs

What function(s) have we looked at that seem similar to exec?

popen() and system()!

How do these differ from exec? Consider this example:

```c
#include <unistd.h>

#include <stdlib.h>

int main(int argc, char**argv) {

  system("ls");

}
```

The system function itself calls fork()! It calls fork, executes the command passed by the parameter, and the original parent process will wait for it to finish.

This means system() is a blocking function -- the parent can not continue until it returns.

The system() function has a lot more overhead but at the same time abstracts the fork-exec-wait pattern for creating processes.

Processes - In C programs

Consider this example, what is its output?

```
int main(int argc, char **argv) {
  pid_t id;
  int status;
  while (--argc && (id=fork())) {
    waitpid(id,&status,0); /* Wait for child*/
  }
  printf("%d:%s\n", argc, argv[argc]);
  return 0;
}
```

Execution:

```
% ./a.out test ing
2:ing
1:test
0:./a.out
```

Processes - In C programs

Here we have the amazing linear time (O(n)) sorting algorithm!

```
int main(int c, char **v) {
        while (--c > 1 && !fork());
        int val  = atoi(v[c]);
        sleep(val);
        printf("%d\n", val);
        return 0;
}
```

Execution:

```
% ./a.out 2 1 8 3 5
1 // took 1 sec
2 // took 2 sec
3 // took 3 sec
5 // took 5 sec
8 // took 8 sec
```

Processes - In C programs

I mentioned how a child process has some differences from the parent process. Here are the key differences:

> The process id and parent id are different between processes
> The parent is notified by a SIGCHLD signal when a child process finishes, the child is not notified when the parent finishes
> The child does not inherit pending signals that were sent to the parent or pending timer alarms

To see all the differences, you can look at the fork man page

Processes - In C programs

```
pid_t children[HELLO_NUMBER];
for(int i = 0; i < HELLO_NUMBER; i++){
    pid_t child = fork();
    if(child == -1)
        break;
    if(child == 0) //I am the child
         execlp("ehco", "echo", "hello", NULL);
     else
         children[i] = child;
 }
 for(int j = 0; j < i; j++){
     waitpid(children[j], NULL, 0);
 }
 return 0;
```

There is a bug somewhere in this code, what is it?

Hint: its a subtle fork bomb.

Processes - In C programs

When a child finishes (or terminates) it still takes up a slot in the kernel process table. Only when the child has been 'waited on' will the slot be available again.

A long running program could create many zombies by continually creating processes and never “wait”-ing for them.

What would be the effect of too many zombies?

> Eventually there would be insufficient space in the kernel process table to create a new processes. Thus fork() would fail and could make the system difficult / impossible to use - for example just logging in requires a new process!

Once a process completes, any of its children will be assigned to "init" - the first process with pid of 1. Thus these children would see getppid() return a value of 1. These orphans will eventually finish and for a brief moment become a zombie. Fortunately, the init process automatically waits for all of its children, thus removing these zombies from the system.

Processes - In C programs

You can prevent zombies from appearing by always waiting on your child like so

```
waitpid(child, &status, 0); // wait for my child process to finish.
```

Note we assume that the only reason to get a SIGCHLD event is that a child has finished (this is not quite true - see man page for more details).

A robust implementation would also check for interrupted status and include the above in a loop. Read on for a discussion of a more robust implementation.

Processes - In C programs

```c
pid_t child;

int main() {

   // Register signal handler BEFORE the child can finish

   signal(SIGCHLD, cleanup); // or better - sigaction

   child = fork();

   if (child == -1) { exit(EXIT_FAILURE);}

   if (child == 0) { /* I am the child!*/

     // Do background stuff e.g. call exec

   } else { /* I'm the parent! */

      sleep(4); // so we can see the cleanup

      puts("Parent is done");

   }

   return 0;

}
```

```c
void cleanup(int signal) {

  int status;

  waitpid(child, &status, 0);

  write(1,"cleanup!\n",9);

}
```

Processes - In C programs

A child's return value is stored in the lowest 8 bits of its exit value.
You can find the lowest 8 bits of the child's exit value (the return value of main() or value included in exit()): Use the "Wait macros" - typically you will use "WIFEXITED" and "WEXITSTATUS" . See wait/waitpid man page for more information).

```
pid_t child = fork();
if (child == -1) return 1; //Failed
if (child > 0) { /* I am the parent - wait for the child to finish */
    pid_t pid = waitpid(child, &status, 0);
    if (pid != -1 && WIFEXITED(status)) {
        int low8bits = WEXITSTATUS(status);
        printf("Process %d returned %d" , pid, low8bits);
    }
} else { /* I am the child */
    execl("/bin/ls", "/bin/ls", ".", (char *) NULL); // "ls ."
}
```

Processes - In C programs

Here is a look at some of these wait macros

```
/* If WIFEXITED(STATUS), the low-order 8 bits of the status. */

#define __WEXITSTATUS(status) (((status) & 0xff00) >> 8)

/* If WIFSIGNALED(STATUS), the terminating signal. */

#define __WTERMSIG(status) ((status) & 0x7f)

/* If WIFSTOPPED(STATUS), the signal that stopped the child. */

#define __WSTOPSIG(status) __WEXITSTATUS(status)

/* Nonzero if STATUS indicates normal termination. */

#define __WIFEXITED(status) (__WTERMSIG(status) == 0)
```

Processes - In C programs

We have talked a lot about signals, a signal is a construct provided to us by the kernel. It allows one process to asynchronously send a signal (think a message) to another process.

If that process wants to accept the signal, it can, and then, for most signals, can decide what to do with that signal. Here is a short list (non comprehensive) of signals. (use man -s7 signal to see more signals)

Name	Default Action	Usual Use Case
SIGINT	Terminate Process (Can be caught)	Tell the process to stop nicely
SIGQUIT	Terminate Process (Can be caught)	Tells the process to stop harshly
SIGSTOP	Stop Process (Cannot be caught)	Stops the process to be continued
SIGCONT	Continues a Process	Continues to run the process
SIGKILL	Terminate Process (Cannot be Ignored)	You want your process gone

Processes - In C programs

We could temporarily pause a child process by sending it a SIGSTOP signal with kill() (equivalent to the kill command). Consider this program:

```
int main() {
  printf("My pid is %d\n", getpid() );
  int i = 60;
  while(--i) {
    write(1, ".",1);
    sleep(1);
  }
  write(1, "Done!",5);
  return 0;
}
```

```
% ./a.out &
My pid is 403
...
% kill -SIGSTOP 403
% kill -SIGCONT 403
```

Lets use the kill command to pause its execution.

Processes - In C programs

We can use the kill system call to do this same thing from C.

```
NAME
       kill - send signal to a process

SYNOPSIS
       #include <sys/types.h>
       #include <signal.h>
       int kill(pid_t pid, int sig);

DESCRIPTION
     The kill() system call can be used to send any signal to any process group
     or process.
```

Processes - In C programs

Recall:

```
int main() {
 printf("My pid is %d\n",
         getpid() );
  int i = 60;
  while(--i) {
    write(1, ".",1);
    sleep(1);
  }
  write(1, "Done!",5);
  return 0;
}
```

Lets write a C program that stops and starts it using kill.

Processes - In C programs

Recall how we used signal before to respond to a SIGCHLD signal. We could also use this to catch a SIGINT signal and gracefully stop our process.

```
int pleaseStop ;
void handle_sigint(int signal) {
  pleaseStop = 1;
}
int main() {
  signal(SIGINT, handle_sigint);
  pleaseStop = 0;
  while ( ! pleaseStop) {
     /* application logic here */
   }
  /* cleanup code here */
```

Note:

`volatile sig_atomic_t pleaseStop;`

Would be better

Processes - In C programs

So, what should you guys worry about for processes?

Know what a process is.

Know the fork-wait-exec pattern and how to write a simple program using it.

Know what a signal is.

Know how we can control processes from the terminal.

Threads

Threads

Not the colloquial term for clothing / stream of posts on a website / a single strand of cloth.

A thread is a similar concept, but entirely different from a process. It is short for “thread of execution.”

A thread represents the sequence of instructions that the CPU has (and will) execute. To remember how to return from function calls, and to store the values of automatic variables and parameters a thread uses a stack.

Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.

The actual system call to create a thread is similar to fork(); it's called clone().

Threads

```
NAME

       clone, __clone2 - create a child process

SYNOPSIS

       #include <sched.h>

       int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, …

         /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );

DESCRIPTION

       clone() creates a new process, in a manner similar to fork(2).


     Unlike fork(2), clone() allows the child process to share parts of its
     execution context with the calling process, such as  the  memory  space,
     the  table  of  file  descriptors, and the table of signal handlers.
```

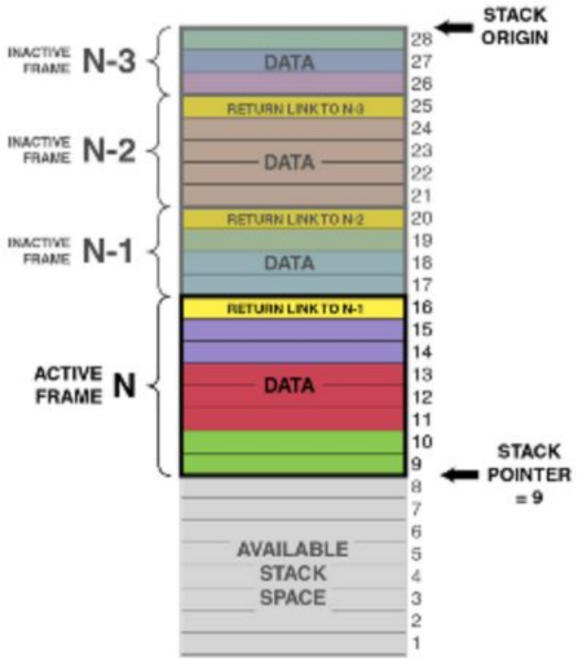
Threads

Your main function (and other functions you might call) has automatic variables. We will store them in memory using a stack and keep track of how large the stack is by using a simple pointer (the "stack pointer").

If the thread calls another function, we move our stack pointer down, so that we have more space for parameters and automatic variables.

Once it returns from a function, we can move the stack pointer back up to its previous value. We keep a copy of the old stack pointer value - on the stack!

This is why returning from a function is very quick - it's easy to 'free' the memory used by automatic variables - we just need to change the stack pointer.

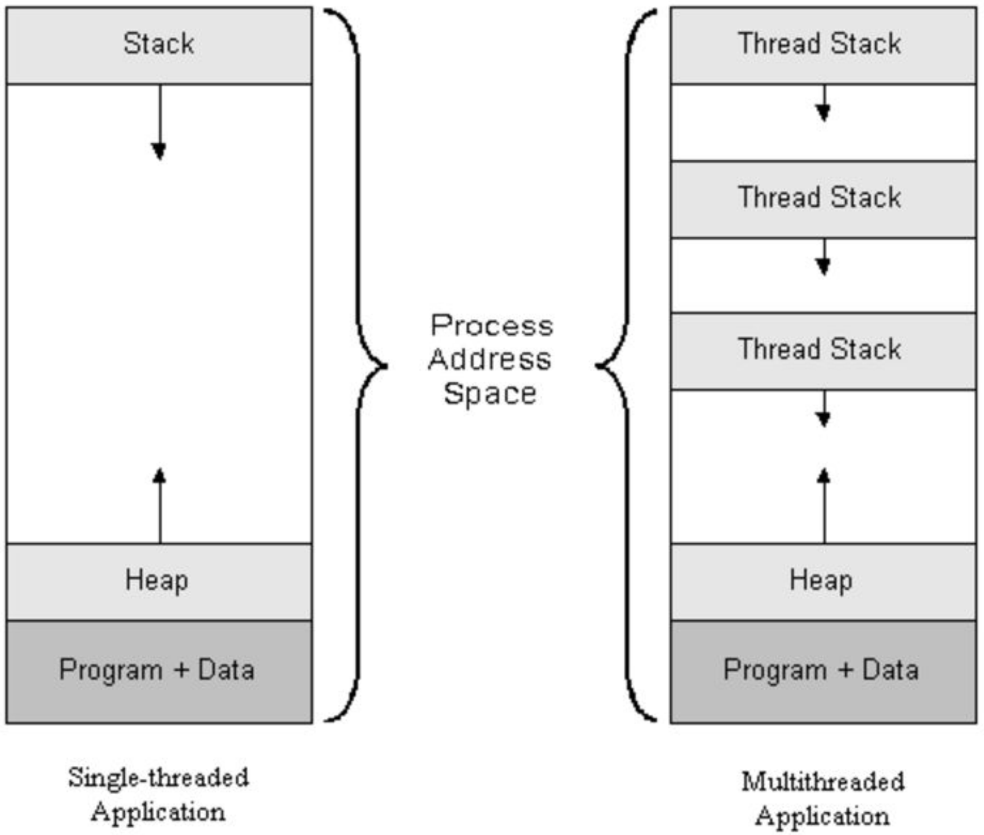


Threads

In a multithreaded program, there are multiple stack but only one address space.

The pthread library allocates some stack space (either in the heap or using a part of the main program's stack) and uses the clone function call to start the thread at that stack address.

The total address space may look something like this.



Threads

You can have more than one thread running inside a process. You get the first thread for free! It runs the code you write inside 'main'. If you need more threads you can call pthread_create to create a new thread using the pthread library.

```
NAME

       pthread_create - create a new thread

SYNOPSIS

       #include <pthread.h>

       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,

                          void *(*start_routine) (void *), void *arg);

       Compile and link with -pthread.

DESCRIPTION

     The  pthread_create() function starts a new thread in the calling process.  The new
     thread starts execution by invoking start_routine(); arg is passed as the sole
     argument of start_routine().
```

Threads

The threads you create all live inside the same virtual memory because they are part of the same process. Thus they can all see the heap, the global variables and the program code etc.

So, you can have two (or more) CPUs working on your program at the same time and inside the same process. It's up to the operating system to assign the threads to CPUs.

If you have more active threads than CPUs then the kernel will assign the thread to a CPU for a short duration (or until it runs out of things to do) and then will automatically switch the CPU to work on another thread.

For example, in a video game, one CPU might be processing the game AI while another thread is computing the graphics output.

Threads

To use pthreads you will need to include `pthread.h` AND you need to compile with `-pthread` (or `-lpthread`) compiler option. This option tells the compiler that your program requires threading support

Recall the prototype for pthread_create:

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

The first is a pointer to a variable that will hold the id of the newly created thread.

The second is a pointer to attributes that we can use to tweak and tune some of the advanced features of pthreads.

The third is a pointer to a function that we want to run

Fourth is a pointer that will be given to our function

Note:

What type is argument 3?

Threads

Here is a simple example, pthread1.c:

```
#include <stdio.h>
#include <pthread.h> // remember to set compilation option -pthread
void *busy(void *ptr) {
    // ptr will point to "Hi"
    puts("Hello World");
    return NULL;
}
int main() {
    pthread_t id;
    pthread_create(&id, NULL, busy, "Hi");
    while (1) {} // Loop forever
}
```

Threads

To wait for a thread to finish, we can use pthread_join().

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

The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.

If retval is not NULL, then pthread_join() copies the exit status of the target thread (i.e., the value that the target thread supplied to pthread_exit(3)) into the location pointed to by *retval.

```
void *result;

pthread_t id;

pthread_create(&id, NULL, busy, "Hi");

pthread_join(id, &result);
```

Threads

Why would we use pthread_join?

Wait for a thread to finish

Clean up thread resources

Grabs the return value of the thread

Finished threads will continue to consume resources. Eventually, if enough threads are created, pthread_create will fail.

In practice, this is only an issue for long-running processes but is not an issue for simple, short-lived processes as all thread resources are automatically freed when the process exits.

Threads

If I call pthread_create twice, how many stacks should my program have?

It should have three -- the original ones plus the additional two for our new threads.

The important idea is that each thread requires a stack because the stack contains automatic variables and the old CPU PC register, so that it can back to executing the calling function after the function is finished.

Remember: threads have unique stacks for function calls and automatic variables, however threads within a process still share the data segment and heap segment of memory. What was stored there?

Threads

Recall exit(): What does this function do?

It completely stops the calling process, all threads included. Remember, exit is equivalent to returning from main.

There is another function, pthread_exit(), that allows us to stop only the thread it is called from.

```
void pthread_exit(void *retval);
```

The pthread_exit() function terminates the calling thread and returns a value via retval that (if the thread is joinable) is available to another thread in the same process that calls pthread_join.

pthread_exit(...) is equivalent to returning from the thread's function; both finish the thread and also set the return value (void *pointer) for the thread.

The pthread library will automatically finish the process if there are no other threads running.

Threads

Calling pthread_exit in the the main thread is a common way for simple programs to ensure that all threads finish. For example, in the following program, the myfunc threads will probably not have time to get started.

```
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, myfunc, "Jabberwocky");
  pthread_create(&tid2, NULL, myfunc, "Vorpel");
  exit(42); //or return 42;
  // No code is run after exit
}
```

Note: this is creating “thread zombies”

Do we care?

Threads

This next program will actually wait for all threads to finish.

```
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, myfunc, "Jabberwocky");
  pthread_create(&tid2, NULL, myfunc, "Vorpel");
  pthread_exit(NULL);

  // No code is run after pthread_exit
  // However process will continue to exist until both threads have finished
}
```

Threads

Alternatively, we join on each thread (i.e. wait for it to finish) before we return from main (or call exit).

```
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, myfunc, "Jabberwocky");
  pthread_create(&tid2, NULL, myfunc, "Vorpel");
  // wait for both threads to finish :
  void* result;
  pthread_join(tid1, &result);
  pthread_join(tid2, &result);
  return 42;
}
```

Threads

There is another function that lets us try to stop a thread

```
int pthread_cancel(pthread_t thread);
```

The pthread_cancel() function sends a cancellation request to the thread thread.

Whether and when the target thread reacts to the cancellation request depends on two attributes that are under the control of that thread: cancelability state and type.

Note the thread may not actually be stopped immediately. For example it can be terminated when the thread makes an operating system call (e.g. write).

In practice, pthread_cancel is rarely used because it does not give a thread an opportunity to clean up after itself (for example, it may have opened some files). An alternative implementation is to use a boolean (int) variable whose value is used to inform other threads that they should finish and clean up.

Threads

Here is an example of usage:

```
void *busy(void *ptr) {
    while (1) { puts("Hello World"); }
    return NULL;
}
int main() {
    pthread_t id;
    pthread_create(&id, NULL, busy, "Hi");
    printf("Canceling...\n");
    pthread_cancel(id);
    printf("Done\n");
}
```

Execution:

```
% ./a.out
Canceling…
Hello world
Hello world
… (about 30 printed out)
Hello world
Done
```

Threads

So, there are several ways to stop a thread from executing

Returning from the thread function

Calling pthread_exit

Cancelling the thread with pthread_cancel

Terminating the process (e.g. SIGTERM); exit(); returning from main

Threads

Both pthread_exit and pthread_join will let the other threads finish on their own (even if called in the main thread).

However, only pthread_join will return to you when the specified thread finishes and let you retrieve the return value.

pthread_exit does not wait and will immediately end your thread and give you no chance to continue executing.

Threads

We are able to pass pointer to stack variables between threads.

Can anyone see an issue with is?

Picture this:

```
pthread_t start_threads() {
  int start = 42;
  pthread_t tid;
  pthread_create(&tid, 0, myfunc, &start); // ERROR!
  return tid;
}
```

Threads

How do we fix that issue?

```
void start_threads() {
  int start = 42;
  void *result;
  pthread_t tid;
  pthread_create(&tid, 0, myfunc, &start); // OK - start will be valid!
  pthread_join(tid, &result);
}
```

Threads - Race Conditions

Here is a joke that describes race conditions:

> A riot is gathering in front of our cpu, the processes are starting to chant…
>
> “What do we want?!”
>
> “Never!”
>
> “When do we want it?!”
>
> “Race conditions!”

Essentially what happened here is we had some job we wanted to do (print out our output), however, one job finished before it was supposed to!

Threads - Race Conditions

The following code is supposed to start ten threads with values 0,1,2,3,...9 However, when run prints out 1 7 8 8 8 8 8 8 8 10! Can you see why?

```
int main() {
    // Each thread gets a
    //different value of i to process
    int i;
    pthread_t tid;
    for(i =0; i < 10; i++) {
        pthread_create(&tid, NULL, myfunc, &i);
    }
    pthread_exit(NULL);
}
```

```
void* myfunc(void* ptr) {
    int i = *((int *) ptr);
    printf("%d ", i);
    return NULL;
}
```

This is a race condition!

Threads - Race Conditions

To overcome this, we would have to give each thread a pointer to its own data. Maybe we could use a struct, and for each thread, store its id, the data we are giving it, and its output, like so:

```
int main() {
    int i, arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    pthread_t tid;
    for(i =0; i < 10; i++) {
        pthread_create(&tid, NULL, myfunc, &arr[i]);
    }
    pthread_exit(NULL);
}
```

```
void* myfunc(void* ptr) {
    int i = *((int *) ptr);
    printf("%d ", i);
    return NULL;
}
```

```
Execution:
     % ./a.out
     3 4 6 1 0 5 2 7 8 9
```

Threads - Race Conditions

The following function is not “thread safe”. Why?

```
char *to_message(int num) {
    char static result [256];
    if (num < 10) sprintf(result, "%d : blah blah" , num);
    else strcpy(result, "Unknown");
    return result;
}
```

Threads - Race Conditions

What is are advantages of using a thread over a process?

> Sharing information between threads is easy because threads (of the same process) live inside the same virtual memory space.
>
> Creating a thread is significantly faster than creating(forking) a process.

What are disadvantages?

> No- isolation! As threads live inside the same process, one thread has access to the same virtual memory as the other threads, meaning security issues could happen.
>
> A single thread can terminate the entire process (e.g. by trying to read address zero).

Threads - Race Conditions

Why would we use a process vs a thread?

Creating separate processes is useful

- When more security is desired (for example, Chrome browser uses different processes for different tabs)
- When running an existing and complete program then a new process is required (e.g. starting 'gcc')
- When you are running into synchronization primitives and each process is operating on something in the system

Threads - reduce

Lets try working through an example using threads of a function that applies a reduction.