Process and signals in linux

Rakesh Basavaraju

Agenda

- ✓ Process Overview
- ✓ Process schedulers
- ✓ Create and destroy
- ✓ Signals

Process Overview

Process

- Process: an instance of a program in execution
- (User) Thread: an execution flow of the process
  - Pthread (POSIX thread) library
- Lightweight process (LWP): used to offer better support for multithreaded applications
  - LWP may share resources: address space, open files, …
  - To associate a lightweight process with each thread
  - Examples of pthread libraries that use LWP: LinuxThreads

- Processes are more than just executing the code also includes resources such as
  - open files and pending signals
  - internal kernel data
  - processor state
  - memory address space with one or more memory mappings
  - one or more thread of execution
  - Data section containing global variables.
- Processes, in effect, are the living result of running program code.

Process

- Each process is represented by a unique identifier, the process ID. The pid is guaranteed to be unique at any single point in time.
- By default, the kernel imposes a maximum process ID value of 32768.
- System administrators can set the value higher via */proc/sys/kernel/pid_max*
- The kernel allocates process IDs to processes in a strictly linear fashion
- The process that spawns a new process is known as the *parent*; the new process is known as the *child*. Every process is spawned from another process.

Process hierarchy

- Every process is spawned from another process.
- Every child has a parent. This relationship is recorded in each process' parent process ID (ppid), which is the pid of the child's parent.
- Each process is owned by a *user* and a *group*.cess.
- This ownership is used to control access rights to resources. To the kernel, users and groups are mere integer values.
- Each process is also part of a *process group*, which simply expresses its relationship to other processes,
- Children normally belong to the same process groups as their parents. In addition, when a shell starts up a pipeline (e.g., when a user enters ls | less), all the commands in the pipeline go into the same process group.

pid

- Pid_t

Programmatically PID is represented by pid_t type defined in <sys/types.h>.

The getpid( ) system call returns the process ID of the invoking process:

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

The getppid( ) system call returns the process ID of the invoking process' parent:

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

Neither call will return an error. Consequently, usage is trivial:

```
printf ("My pid=%d\n", getpid ( ));
printf ("Parent's pid=%d\n", getppid ( ));
```

Process states

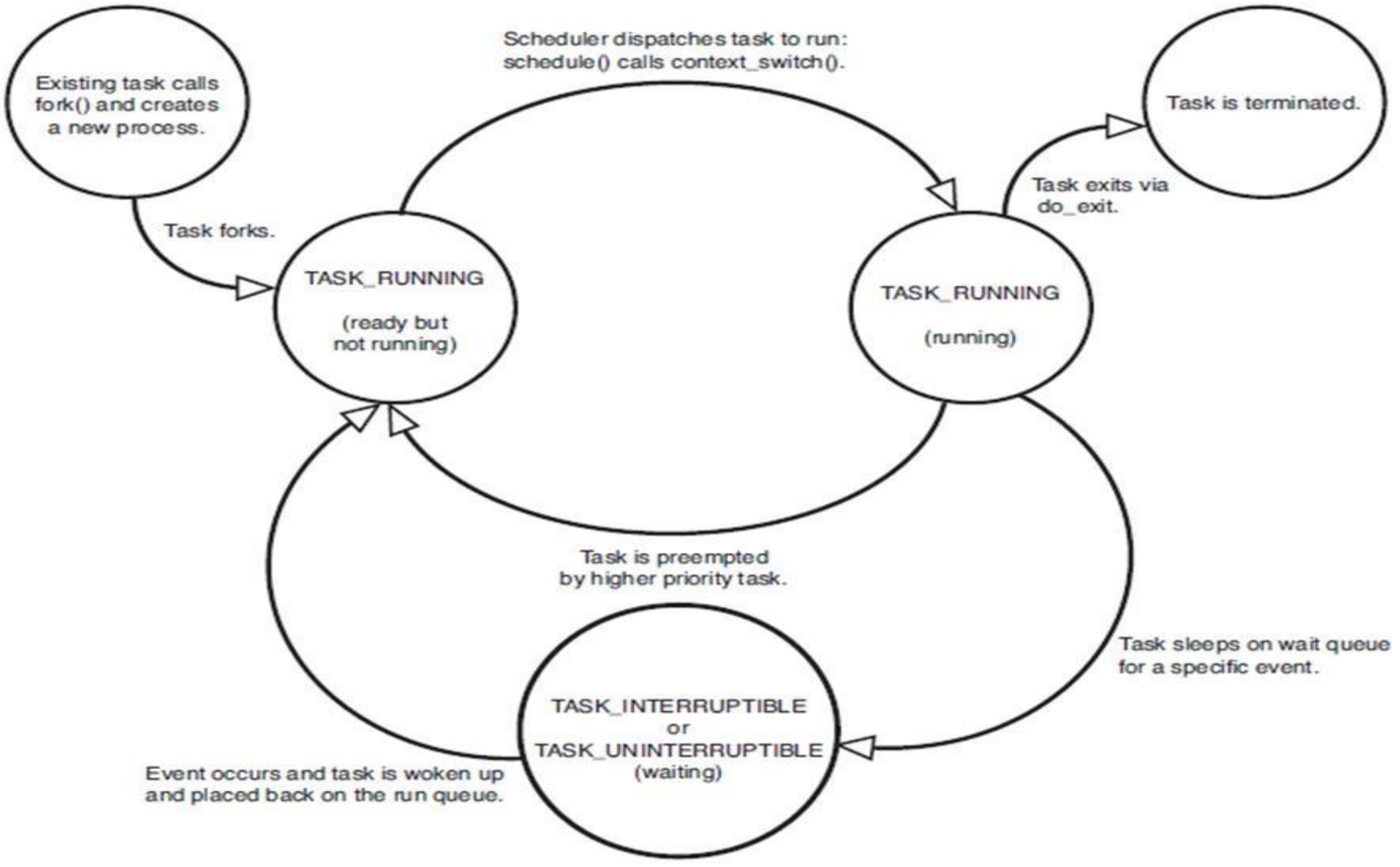

Existing task calls fork() and creates a new process.
Task forks.
Scheduler dispatches task to run: schedule() calls context_switch().
Task is terminated.
Task exits via do_exit.
TASK_RUNNING (ready but not running)
TASK_RUNNING (running)
Task is preempted by higher priority task.
Task sleeps on wait queue for a specific event.
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
Event occurs and task is woken up and placed back on the run queue.

Figure 3.3 Flow chart of process states.

Process priority

- Linux kernel implements two separate priority ranges.
  1. Nice value
  2. Real-time priority

- Nice: a number from -20 to +19 with a default of 0.

Larger value corresponds to lower priority. (You are being "nice" to other processes)

Lesser values correspond to higher priority.

- Real-time priority: Values are configurable, but by default range is from 0 to 99, inclusive.

Higher priority values corresponds to higher priority,

ps -eo state,uid,pid,ppid,rtprio,time,comm.

A value of "-" means the process is not real-time.

- To launch a programe with your required priority using

nice -n nice_value program_name

- To change the priority of an already running process using

renice -n nice_value -p process_id

Process priority

- Linux provides several system calls for retrieving and setting a process's nice value.

```
#include<unistd.h>
Int nice(int inc); // On success nice value will be incremented by inc and returns new updated value.
//On error returns -1
Since -1 is also legal value, to determine success and failure you can do this

int ret;
errno=0;
ret=nice(10);
If(ret==-1 && errno!=0)
   perror(“nice”);
else
   printf (“nice value is now %d\n”,ret);
```

Process Descriptor

- task_struct data structure
  - state: process state
  - thread_info: low-level information for the process
  - mm: pointers to memory area descriptors
  - tty: tty associated with the process
  - fs: current directory
  - files: pointers to file descriptors
  - signal: signals received
  - …
- The *task_struct* has several sub-structures that it references:

  tty_struct – TTY associated with the process

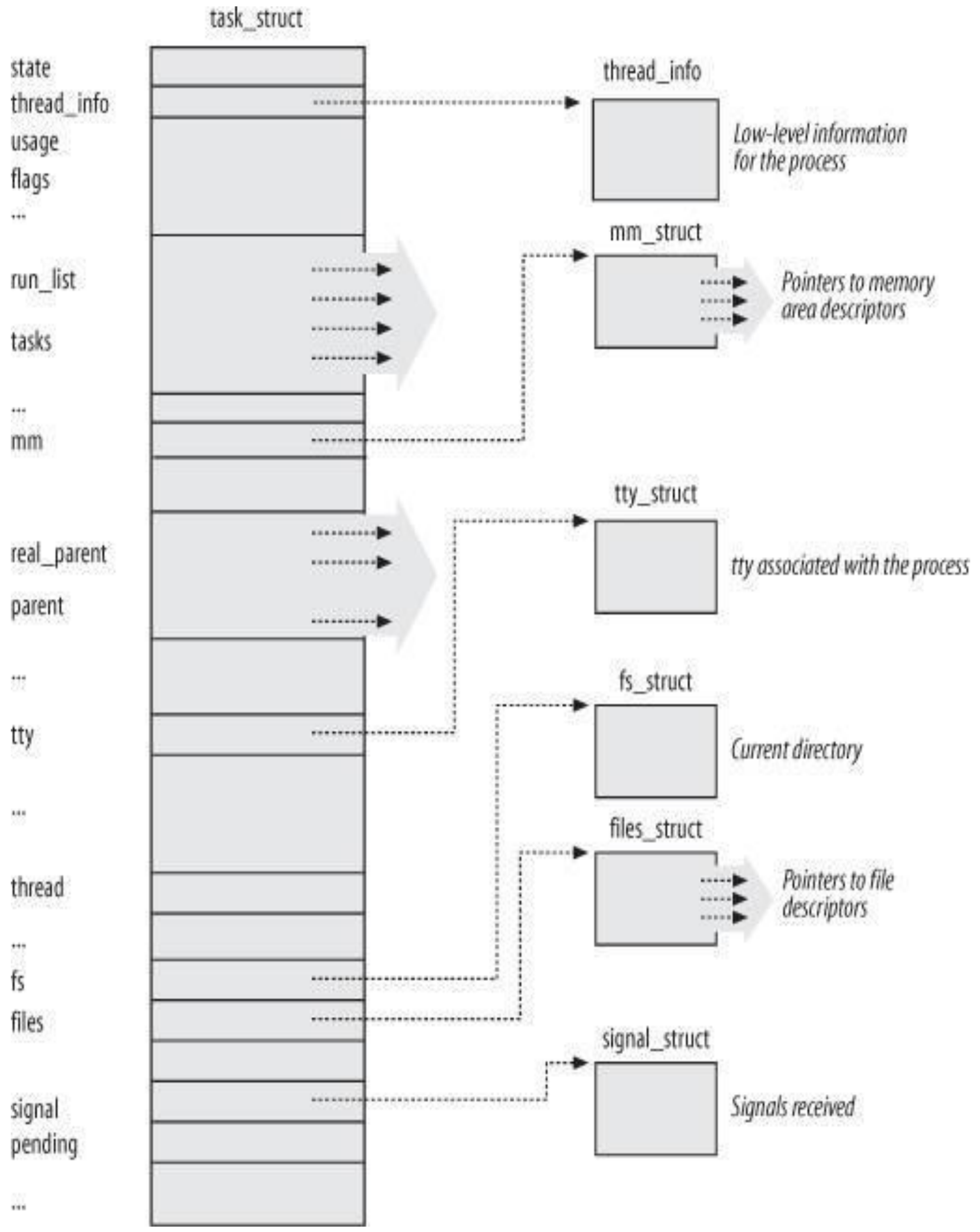
  fs_struct – current and root directories associated with the process

  files_struct – file descriptors for the process

  mm_struct – memory areas for the process

  signal_struct – signal structures associated with the process

  user_struct – per-user information (for example, number of current processes)



Task list

- The kernel stores the list of process in a circular doubly linked list called task_list. Each element in the task_list is a process descriptor of the type struct task_struct defined in <linux/sched.h>

- Process descriptor contains the data that describes the execution program – open files, process's address space, pending signals, process's state and much more.

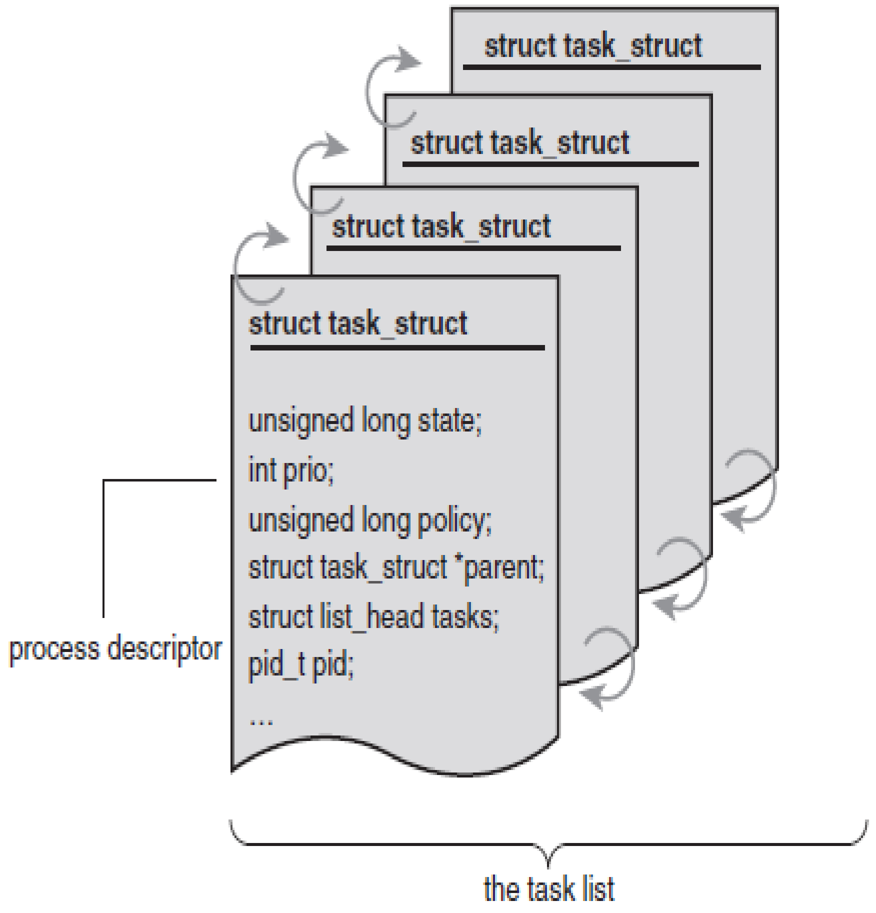


Figure 3.1 The process descriptor and task list.

Process schedulers

Process scheduling

- The process scheduler decides which process runs, when, and for how long.
- The process scheduler (or simply the *scheduler*, to which it is often shortened) divides the finite resource of processor time between the runnable processes on a system.
- The scheduler is the basis of a multitasking operating system such as Linux.
- By deciding which process runs next, the scheduler is responsible for best utilizing the system and giving users the impression that multiple processes are executing simultaneously.
- A pre-processor macro from the header <sched.h> represents each policy: the macros are SCHED_FIFO, SCHED_RR, and SCHED_OTHER.

- Different schedulers will have different goals
  - Maximize throughput
  - Minimize latency
  - Prevent indefinite postponement
  - Complete process by given deadline
  - Maximize processor utilization

preemptive and non-preemptive scheduling

Preemptive processes

- Can be removed from their current processor
- Can lead to improved response times
- Important for interactive environments
- Preempted processes remain in memory

Nonpreemptive processes

- Run until completion or until they yield control of a processor
- Unimportant processes can block important ones indefinitely

Scheduling

- Timeslice

The timeslice that Linux allots to each process is an important variable in the overall behavior and performance of a system.

If timeslices are too large, processes must wait a long time in between executions

If the timeslices are too small, a significant amount of the system's time is spent switching from one application to another

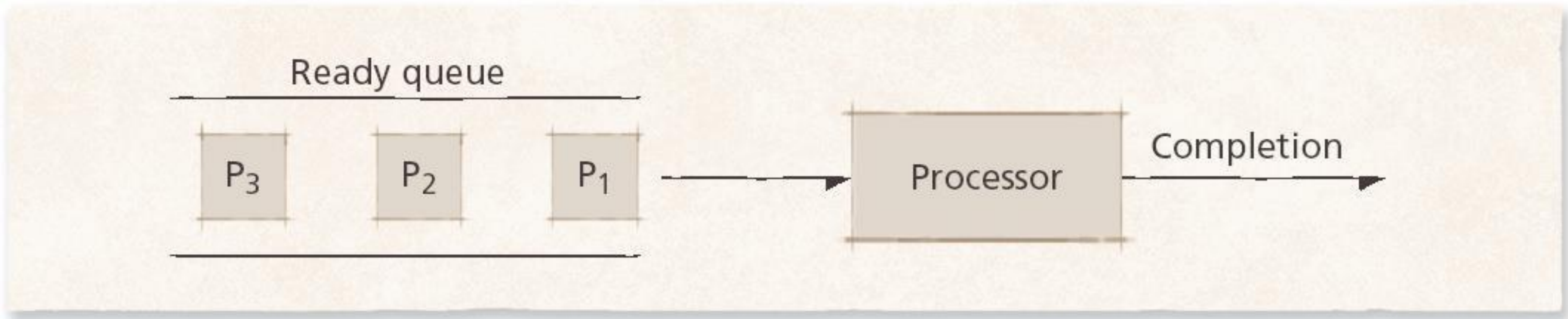
- I/O- Versus Processor-Bound Processes

Processes that continually consume all of their available timeslices are considered processor-bound.

Processes that spend more time blocked waiting for some resource than executing are considered I/O-bound.

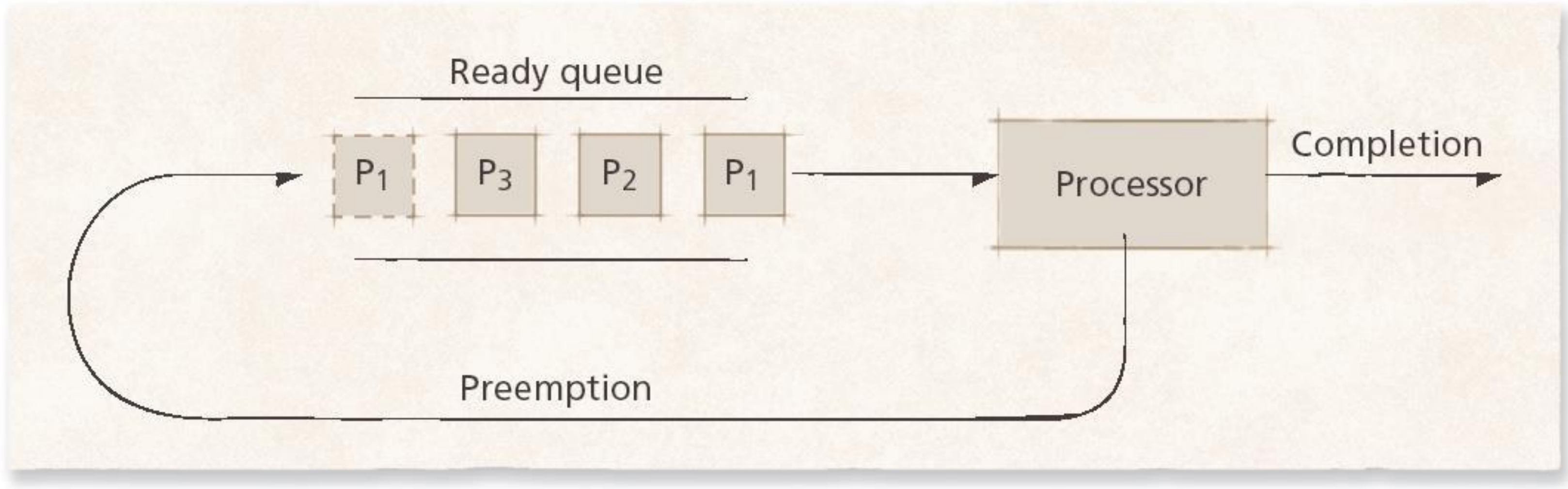
First in, First out

- FIFO-classed process will continue running as long as no higher-priority process becomes runnable.
- FIFO-classed process will always run if it is the highest-process on the system.
- It will continue to run until it blocks or calls sched_yield(), or higher-priority process becomes runnable.
- When FIFO-classed process blocks, the scheduler removes it from the list of the runnable process. When it again becomes runnable again it is inserted at the end of list of process at its priority. Thus it will not run until any other process of higher or equal priority ceases execution.



Round Robin

- Scheduler assigns each RR –classed process a timeslice. When an RR-classed process exhausts its timeslice, the scheduler moves it to the end of the list of process at its priority.
- Processes run only for a limited amount of time called a time slice or quantum
- Preemptible
- Requires the system to maintain several processes in memory to minimize overhead
- Often used as part of more complex algorithms



Normal policy and batch scheduling policy

- The normal policy

SCHED_OTHER represents the standard scheduling policy, the default nonreal-time class. All normal-classed processes have a static priority of 0. Consequently, any runnable FIFO- or RR-classed process will preempt a running normal-classed process.

- The batch scheduling policy

SCHED_BATCH is the *batch* or *idle scheduling policy*. Processes in this class run only when there are no other runnable processes on the system, even if the other processes have exhausted their timeslices.

Create and destroy

Create and destroy

- The act of loading into memory and executing a program image is separate from the act of creating a new process. One system call loads a binary program into memory, replacing the previous contents of the address space, and begins execution of the new program.

- The act of creating a new process is called forking, and this functionality is provided by the fork( ) system call. Two acts—first a fork, to create a new process, and then an exec, to load a new image into that process—are thus required to execute a new program image in a new process.

fork

- A new process running the same image as the current one can be created via the fork( )

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

In the child, a successful invocation of fork( ) returns 0. In the parent,fork( ) returns the pid of the child.

On error, a child process is not created, fork( ) returns -1, and errno is set as below

EAGAIN- The kernel failed to allocate certain resources, such as a new pid, or theRLIMIT_NPROC resource limit (rlimit) has been reached .

ENOMEM- Insufficient kernel memory was available to complete the request.

fork

- The kernel created copies of all internal data structures
- duplicated the process' page table entries, and then performed a page-by-page copy of the parent's address space into the child's new address space. But this page-by-page copy is time-consuming.
- Instead of a wholesale copy of the parent's address space, modern Unix systems such as Linux employ *copy-on-write*(COW) pages.

Copy on Write:

- parent and child read the same physical pages
- Lightweight process: parent and child share per-process kernel data structures
- vfork() system call: parent and child share the memory address space

exec

To load a new image into that process—are thus required to execute a new program image in a new process.

There is family of exec system call

int execl (const char *path, const char *arg, ...);

execl replaces the current process image with a new one by loading into memory the program pointed at by path. The parameter arg is the first argument to this program.

execl( ) call changes address space and process image and other attributes of the process:

- Any pending signals are lost.
- Any memory locks are dropped.
- Most thread attributes are returned to the default values.
- Most process statistics are reset.
- Anything related to the process' memory, including any mapped files, is dropped.
- Anything that exists solely in user space

Exec family

- int execlp (const char *file, const char *arg, ...);
- int execle (const char *path, const char *arg, ..., char * const envp[]);
- int execv (const char*path, char *const argv[]);
- int execvp (const char *file, char *constargv[]);
- int execve (const char *filename, char *const argv[], char *const envp[]);

l – list

v – array (vector)

- The l and v delineate whether the arguments are provided via a *l*ist or an array (*v*ector).

p – user full path is searched for the given file

e- new environment is also supplied for the new process

Terminating a Process

- A call to exit( ) performs some basic shutdown steps, and then instructs the kernel to terminate the process.

  ```
  #include <stdlib.h>
  void exit (int status);
  ```

  Before terminating the process, the C library performs the following shutdown steps, in order:

  1. Call any functions registered with atexit( ) or on_exit( ), in the reverse order of their registration.
  2. Flush all open standard I/O streams.
  3. Remove any temporary files created with the tmpfile( ) function.

  exit( ) invokes the system call _exit( ) to let the kernel handle the rest of the termination process:

  ```
  #include <unistd.h>
  void _exit(int status)
  ```

  When a process exits, the kernel cleans up all of the resources that it created on the process' behalf that are no longer in use. This includes, allocated memory, open files, and System V semaphores.
  After cleanup, kernel destroys the process and notifies the parent of its child's demise.

- A process can also terminate if it is sent a signal whose default action is to terminate the process. Such signals include SIGTERM and SIGKILL.

atexit()

- Linux implements, the atexit( ) library call, used to register functions to be invoked on process termination. A successful invocation of atexit( ) registers the given function to run during normal process termination; i.e., when a process is terminated via either exit( ) or a return from main( ).

If a process terminates via a signal, the registered functions are not called.

```
#include <stdlib.h>

int atexit (void (*function)(void));
```

On success, atexit( ) returns 0. On error, it returns -1.

```
#include <stdio.h>
#include <stdlib.h>
void out (void)
{
   printf ("atexit( ) succeeded!\n");
}
int main (void)
{
        if (atexit (out))
                        fprintf(stderr, "atexit( ) failed!\n");
        return 0;
}
```

Waiting for terminated child process

- If a child process were to entirely disappear when terminated, as one might expect, no remnants would remain for the parent to investigate.

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

A call to wait( ) returns the pid of a terminated child, or -1 on error.

- If no child has terminated, the call blocks until a child terminates. If a child has already terminated,

the call returns immediately.

On error, there are two possible errno values:

ECHILD

The calling process does not have any children.

EINTR

A signal was received while waiting, and the call returned early.

wait

- If not NULL, the status pointer contains additional information about the child.

```
#include <sys/wait.h>
int WIFEXITED (status);
int WIFSIGNALED (status);
int WIFSTOPPED (status);
int WIFCONTINUED (status);
int WEXITSTATUS (status);
int WTERMSIG (status);
int WSTOPSIG (status);
int WCOREDUMP (status);
```

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
int main (void)
{
   int   status;
   pid_t   pid;
   if (!fork ( ))
           return 1;
   pid = wait (&status);
   if (pid == -1)
           perror ("wait");
   printf ("pid=%d\n", pid);
   if (WIFEXITED (status))
       printf ("Normal termination with exit status=%d\n",WEXITSTATUS (status));
   if (WIFSIGNALED (status))
       printf ("Killed by signal=%d%s\n", WTERMSIG (status), WCOREDUMP (status) ? "  (dumped
core)" : "");
   if (WIFSTOPPED (status))
       printf ("Stopped by signal=%d\n", WSTOPSIG (status));
   if (WIFCONTINUED (status))
       printf ("Continued\n");
   return 0;
}
```

```
$ ./wait
pid=8529
Normal termination with exit status=1
```

```
$ ./wait
pid=8678
Killed by signal=6
```

waitpid

- Assume a process has multiple children, and does not wish to wait for all of them, but rather for a specific child process ??
- If you know the pid of the process you want to wait for, you can use the waitpid( ) system call:

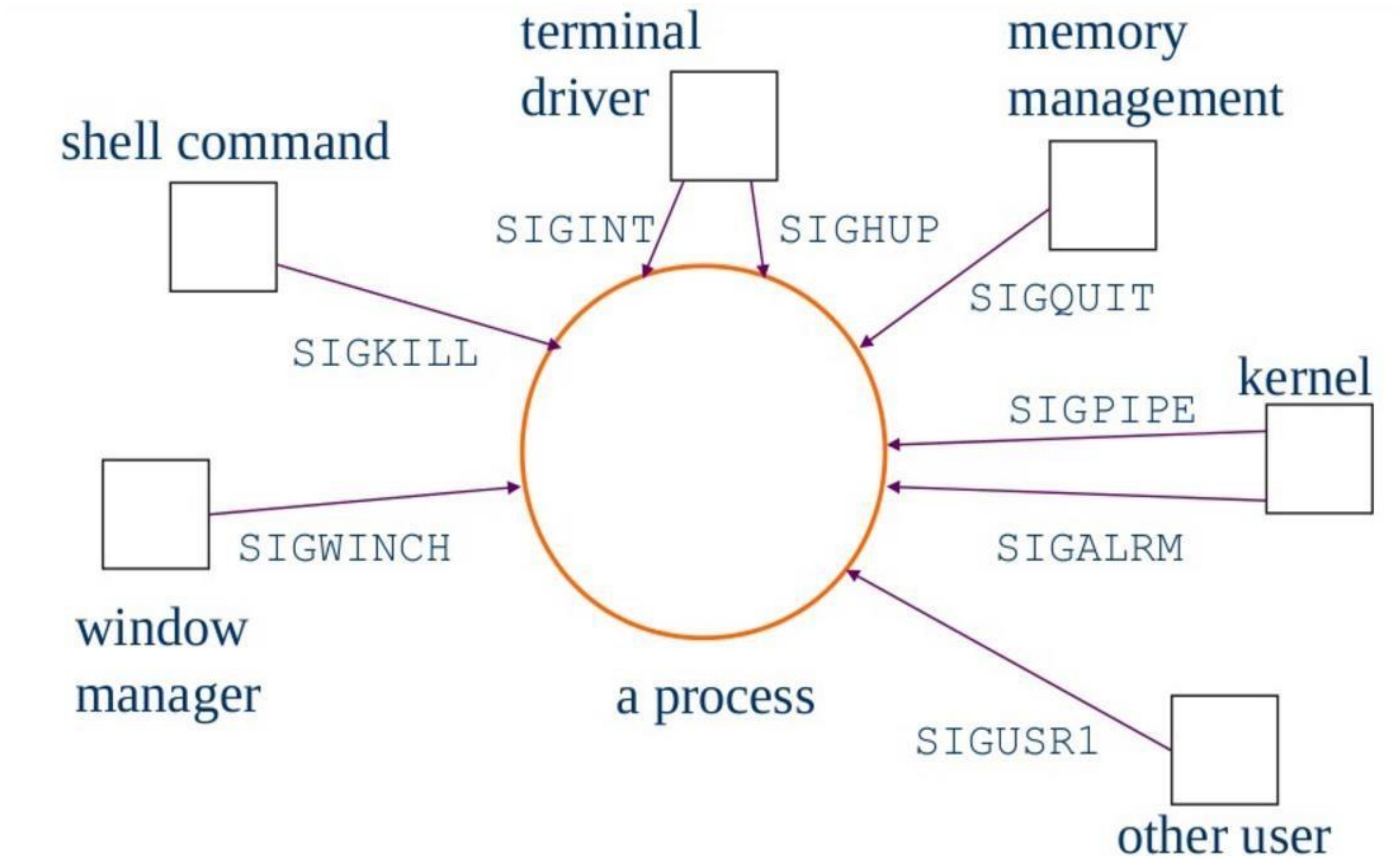
```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int *status, int options);
On success, waitpid( ) returns the pid of the process whose state has changed
On error, the call returns -1

int status;
pid_t pid;
pid = waitpid (1742, &status, WNOHANG);
if (pid == -1)
   perror ("waitpid");
else {
   printf ("pid=%d\n", pid);
   if (WIFEXITED (status))
   printf ("Normal termination with exit status=%d\n", WEXITSTATUS (status));
if (WIFSIGNALED (status))
   printf ("Killed by signal=%d%s\n", WTERMSIG (status), WCOREDUMP (status) ? " (dumped core)" :
"");
}
```

Signals

signals

- A signal is an asynchronous event which is delivered to a process.
- Asynchronous means that the event can occur at any time may be unrelated to the execution of the process.
- Signals are raised by some error conditions, such as memory segment violations, floating point processor errors, or illegal instructions. – e.g. user types ctrl-C, or the modem hangs

Signal Sources



POSIX Defined signal

- SIGALRM: Alarm timer time-out. Generated by alarm( ) API.
- SIGABRT: Abort process execution. Generated by abort( ) API.
- SIGFPE: Illegal mathematical operation.
- SIGHUP: Controlling terminal hang-up.
- SIGILL: Execution of an illegal machine instruction.
- SIGINT: Process interruption. Can be generated by <Delete> or <ctrl_C> keys.
- SIGKILL: Sure kill a process. Can be generated by

  "kill -9 <process_id>" command.
- SIGPIPE: Illegal write to a pipe.
- SIGQUIT: Process quit. Generated by <crtl_\> keys.
- SIGSEGV: Segmentation fault. generated by de-referencing a NULL pointer.

POSIX Defined signal

- SIGTERM: process termination. Can be generated by "kill <process_id>" command.
- SIGUSR1: Reserved to be defined by user.
- SIGUSR2: Reserved to be defined by user.
- SIGCHLD: Sent to a parent process when its child process has terminated.
- SIGCONT: Resume execution of a stopped process.
- SIGSTOP: Stop a process execution.
- SIGTTIN: Stop a background process when it tries to read from from its controlling terminal.
- SIGTSTP: Stop a process execution by the control_Z keys.
- SIGTTOUT: Stop a background process when it tries to write to its controlling terminal.

Actions on signals

Process that receives a signal can take one of three action:

1. Perform the system-specified default for the signal – notify the parent process that it is terminating; – generate a core file; (a file containing the current memory image of the process) – terminate.
2. Ignore the signal – A process can do ignoring with all signal but two special signals: SIGSTOP and SIGKILL.
3. Catch the Signal – When a process catches a signal, except SIGSTOP and SIGKILL, it invokes a special signal handing routine.

Signals supported by Linux

SIGNAL	DESCRIPTION
SIGABRT	The abort( ) function sends this signal to the process that invokes it. The process then terminates and generates a core file. In Linux, assertions such as assert( ) call abort( ) when the conditional fails.
SIGALRM	The alarm( ) and setitimer( ) (with the ITIMER_REAL flag) functions send this signal to the process that invoked them when an alarm expires.
SIGBUS	The kernel raises this signal when the process incurs a hardware fault other than memory protection, which generates a SIGSEGV.
SIGCHLD	Whenever a process terminates or stops, the kernel sends this signal to the process' parent. Because SIGCHLD is ignored by default, processes must explicitly catch and handle it if they are interested in the lives of their children.
SIGCONT	The kernel sends this signal to a process when the process is resumed after being stopped.
SIGILL	The kernel sends this signal when a process attempts to execute an illegal machine instruction. The default action is to terminate the process, and generate a core dump.

Signals supported by Linux

SIGNAL	DESCRIPTION
SIGINT	This signal is sent to all processes in the foreground process group when the user enters the interrupt character (usually Ctrl-C). The default behavior is to terminate;
SIGKILL	This signal is sent from the kill( ) system call;
SIGPIPE	If a process writes to a pipe, but the reader has terminated, the kernel raises this signal. The default action is to terminate the process, but this signal may be caught and handled.
SIGSEGV	This signal, whose name derives from *segmentation violation*, is sent to a process when it attempts an invalid memory access.
SIGSTOP	This signal is sent only by kill( ). It unconditionally stops a process, and cannot be caught or ignored.
SIGUSR1 *and* SIGUSR2	These signals are available for user-defined purposes; the kernel never raises them. Processes may use SIGUSR1 and SIGUSR2 for whatever purpose they like. A common use is to instruct a daemon process to behave differently. The default action is to terminate the process.

Sample program

a signal handler for SIGINT that simply prints a message and then terminates the program (as SIGINT would do anyway)

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
/* handler for SIGINT */
static void sigint_handler (int signo)
{
        printf ("Caught SIGINT!\n");
        exit (EXIT_SUCCESS);
}
int main (void)
{
/* Register sigint_handler as our signal handler for SIGINT. */  if
(signal (SIGINT, sigint_handler) == SIG_ERR) {
        fprintf (stderr, "Cannot handle SIGINT!\n");
        exit (EXIT_FAILURE);
}
for (;;)
        pause ( );
return 0;
}
```

kill

kill Command

–kill signal pid •

Send a signal of type signal to the process with id pid

Can specify either signal type name (-SIGINT) or number (-2)

Examples

–kill –2 1234

–kill SIGINT 1234

Same as pressing Ctrl-c if process 1234 is running in foreground Send signals via commands

Signal and sigaction

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal (int signo, sighandler_t handler);
```

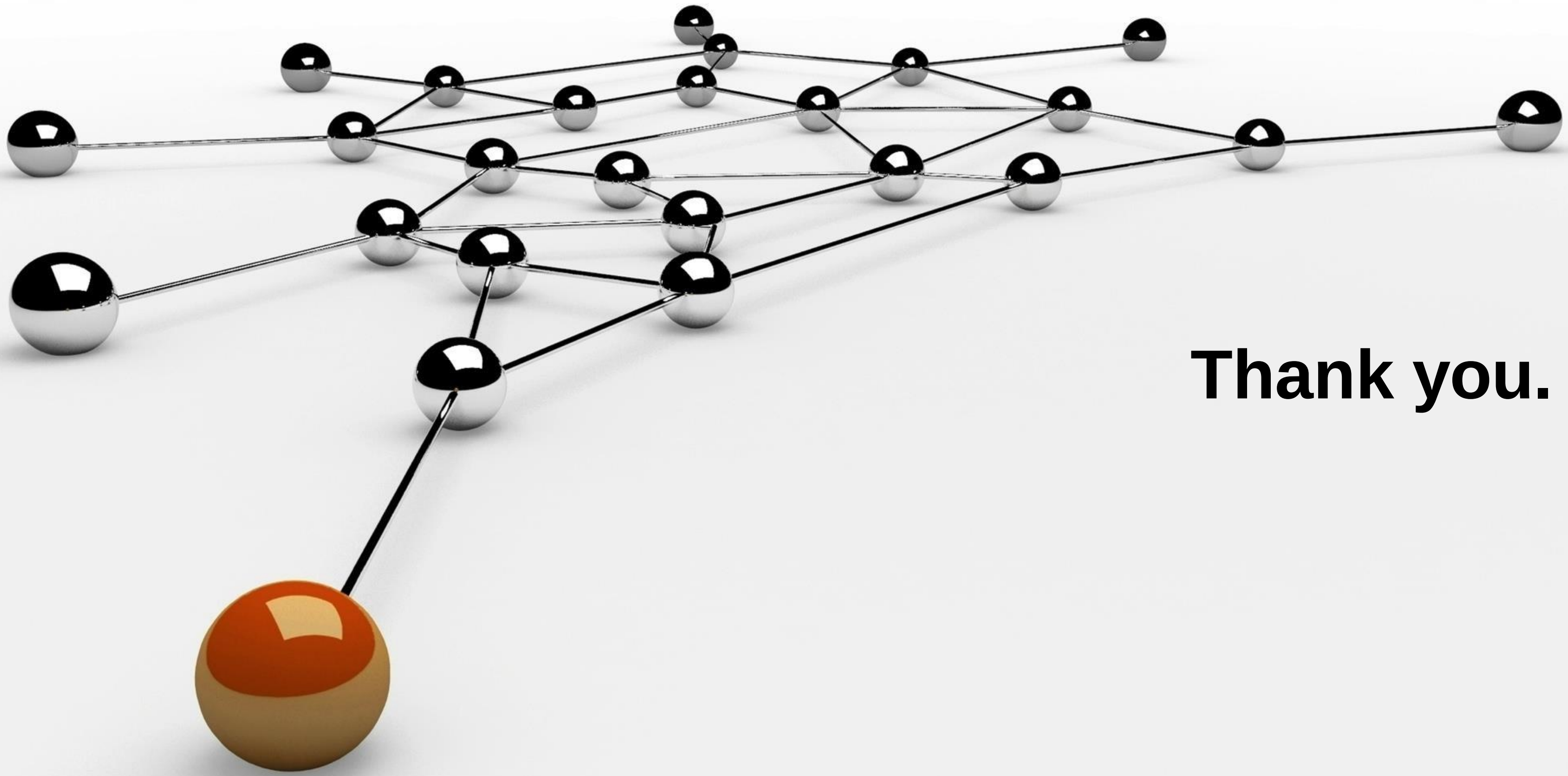
A successful call to signal( ) removes the current action taken on receipt of the signal signo, and instead handles the signal with the signal handler specified by handler.

```
int sigaction (int signo, const struct sigaction *act, struct sigaction *oldact);
```

- sigaction( ) changes the behavior of the signal identified by signo,
- If act is not NULL, the system call changes the current behavior of the signal as specified by act.
- If oldact is not NULL, the call stores the previous (or current, if act is NULL) behavior of the given signal there.

Aricent®

Engineering excellence. Sourced.



Thank you.

www.aricent.com