Práctica 7: Procesos

Esta práctica se ha realizado en una máquina virtual corriendo el sistema operativo Ubuntu 15.04 LTS.

Apartado 1: Políticas de planificación

***Ejercicio 1.*** El planificador y prioridad de un proceso puede consultarse con el comando chrt. Adicionalmente el comando nice/renice permite ajustar la prioridad del planificador. Consultar la página de manual de ambos comandos y comprobar su funcionamiento cambiando el “nice” de la shell a -10 y su política de planificación a FIFO con prioridad 12.

El comando chrt manipula los atributos de un proceso en tiempo real.

Set policy:

chrt [options] [<policy>] [-p <priority> <pid> | <command> [<arg>...]]  
Get policy:  
 chrt [options] -p <pid>

Options:  
 -a | --all-tasks operate on all the tasks (threads) for a given pid  
 -h | --help display this help  
 -m | --max show min and max valid priorities  
 -p | --pid operate on existing given pid  
 -v | --verbose display status information  
 -V | --version output version information

Scheduling policies:  
 -b | --batch set policy to SCHED\_BATCH  
 -f | --fifo set policy to SCHED\_FIFO  
 -i | --idle set policy to SCHED\_IDLE  
 -o | --other set policy to SCHED\_OTHER  
 -r | --rr set policy to SCHED\_RR (default)

Ejemplo de Get Policy:

chrt -a -p 2724 (En este caso, 2724 es el pid del proceso del bash).  
- pid 2724's current scheduling policy: SCHED\_OTHER  
- pid 2724's current scheduling priority: 0`

El comando nice sirve para asignar prioridades a comandos (tareas) ejecutados desde el terminal. Si no hay comando, muestra el valor de `nice' actual. El rango de valores de `nice' abarca desde -20 (más favorable al proceso) hasta 19 (menos favorable al proceso).

nice [OPTION] [COMMAND [ARG]...]

-n, --adjustment=N add integer N to the niceness (default 10)

Example: nice -n10 /bin/bash/

Para ver la prioridad del proceso, hacer ps -al. NI es la prioridad de la tarea en el planificador. PRI es la prioridad actual del proceso.

Renice tiene el mismo funcionamiento que nice, pero cambia la prioridad de la tarea una vez ejecutada en el proceso de la shell.   
renice [-n] <priority> [-p|--pid] <pid>...  
 renice [-n] <priority> -g|--pgrp <pgid>...  
 renice [-n] <priority> -u|--user <user>...

Cambiar el “nice” de la shell a -10 y su política de planificación a FIFO con prioridad 12.

renice -n -10 -p 2724

chrt -f -p 12 2724

***Ejercicio 2.*** Escribir un programa que muestre la política de planificación en una representación en cadena y la prioridad (igual que chrt). Además de mostrar los valores máximo y mínimo de la prioridad para la política de planificación.

**#include <sched.h>**

**#include <sys/types.h>**

**#include <stdio.h>**

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

**int policy;**

**struct sched\_param params;**

**policy = sched\_getscheduler(0);**

**sched\_getparam(0, &params);**

**int max\_priority = sched\_get\_priority\_max(policy);**

**int min\_priority = sched\_get\_priority\_min(policy);**

**printf("Proceso actual: %i\n",getpid());**

**if(policy == 0) {**

**printf("Política del proceso: SCHED\_OTHER\nPrioridad: %i\n",params.sched\_priority);**

**} else if(policy == 1) {**

**printf("Política del proceso: SCHED\_FIFO\nPrioridad: %i\n",params.sched\_priority);**

**} else if(policy == 2) {**

**printf("Política del proceso: SCHED\_RR\nPrioridad: %i\n",params.sched\_priority);**

**}**

**printf("Prioridad máxima: %i Prioridad minima: %i\n", max\_priority, min\_priority);**

**return 0;**

**}**

***Ejercicio 3.*** Ejecutar el programa anterior en una shell con prioridad 12 y política de planificación SCHED\_FIFO como la del ejercicio 1. ¿Cuál es la prioridad en este caso del programa? ¿Se heredan los atributos de planificación?

La prioridad del programa en este caso del programa es 12. Si, se heredan los atributos de planificación.

Apartado 2: Grupos de Procesos y Sesiones. Recursos de un proceso.

***Ejercicio 1.*** El comando ps es de especial importancia para ver los procesos del sistema y su estado. Estudiar la página de manual

El comando ps hace un reporte de los procesos actuales.

ps [options]

Esta versión de ps acepta distintos tipos de opciones:

1. Opciones de UNIX, las cuales pueden ser agrupadas y deben estar precedidas por un guión.

2. Opciones BDS, las cuales deben estar agrupadas y no se suele usar con guiones.

3. Opciones largas GNU, las cuales están precedidas por dos guiones.

ps --ayuda

● Mostrar todos los procesos del usuario actual en formato extendido

Formatos de salida:

-F extra full

-f full-format, including command lines

-l long format

● Mostrar los procesos del sistema, incluyendo el identificador del proceso, del grupo, la sesión, el estado y la línea de comandos.

ps -o pid,pgid,sess,state,cmd

● Observar el identificador de proceso, grupo y sesión de los procesos. ¿Qué identificadores comparten la shell y los programas que se ejecutan en ella?¿Cuál es el identificador de grupo de procesos cuando se crea un nuevo proceso?

ps -o pid,pgid,sess

Comparten el identificador de sesión. El identificador de grupo de procesos cuando se crea un nuevo proceso es el mismo que el del proceso.

***Ejercicio 2.*** Escribir un programa que muestre los identificadores de un proceso: identificador, identificador de grupo y de sesión. Mostrar además el número de archivos que puede abrir el proceso y el directorio de trabajo actual.

**#include <unistd.h>**

**#include <sys/types.h>**

**#include <stdio.h>**

**#include <sys/time.h>**

**#include <sys/resource.h>**

**#include <stdlib.h>**

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

**/\*if (argc != 2) {**

**fprintf(stderr, "Usage: %s <pid>\n", argv[0]);**

**exit(EXIT\_FAILURE);**

**}\*/**

**pid\_t pid = getpid();**

**pid\_t pgid = getpgid(pid);**

**pid\_t ses\_id = getsid(pid);**

**char dir[1024];**

**struct rlimit rlim;**

**getrlimit(RLIMIT\_NOFILE, &rlim);**

**getcwd(dir, sizeof(dir));**

**printf("ID del proceso: %i \n", pid);**

**printf("PGID del proceso: %i \n", pgid);**

**printf("Identificador de sesión %i \n", ses\_id);**

**printf("Número máximo de ficheros que puede abrir: %lld\n",(long long int)rlim.rlim\_max);**

**printf("Directorio de trabajo: %s \n", dir);**

**return 0;**

**}**

***Ejercicio 3.*** Normalmente un demonio está en su propia sesión y grupo. Para garantizar que es posible crear la sesión y grupo el proceso hace un fork() en el que ejecuta la lógica del demonio y crea la nueva sesión. Escribir una plantilla de demonio (fork y la creación de una nueva sesión) en el que únicamente se muestren los atributos de los procesos (como en el ejercicio 2). Además un demonio tiene un directorio de trabajo definido, fijar el de nuestra plantilla al /tmp.

¿Qué sucede si el proceso padre termina antes de que el hijo imprima su información (observar el ID del proceso padre)?¿Y si el proceso que termina antes es el hijo (observar el estado del proceso hijo)?

Pues que el hijo se queda huerfano, entonces el proceso init hereda al hijo. Si el hijo termina antes que el padre se convierte en zombie.

**#include <sys/types.h>**

**#include <unistd.h>**

**#include <sys/time.h>**

**#include <sys/resource.h>**

**#include <stdio.h>**

**#include <errno.h>**

**#include <sys/types.h>**

**#include <sys/wait.h>**

**#include <stdlib.h>**

**int showProcessInfo(char \*id) {**

**if (id == NULL)**

**return -1;**

**pid\_t pid = getpid();**

**gid\_t gid = getgid();**

**pid\_t sid = getsid(pid);**

**struct rlimit fileLimit;**

**int rc = getrlimit(RLIMIT\_NOFILE, &fileLimit);**

**if (rc == -1) {**

**perror("Unable to get the resource limits");**

**return -1;**

**}**

**char \*path = malloc(sizeof(char)\*(4096 + 1));**

**char \*rpath = getcwd(path, 4096 + 1);**

**if (rpath == NULL) {**

**perror("Unable to get the CWD path");**

**free(path);**

**return -1;**

**}**

**printf("[%s] PID: %i\n", id, pid);**

**printf("[%s] GID: %i\n", id, gid);**

**printf("[%s] SID: %i\n", id, sid);**

**printf("[%s] Max num of files: %ld\n", id, fileLimit.rlim\_max);**

**printf("[%s] CWD: %s\n", id, path);**

**free (path);**

**return 0;**

**}**

**int main() {**

**pid\_t pid = fork();**

**pid\_t nsid;**

**switch (pid) {**

**case -1:**

**perror("Unable to fork");**

**return -1;**

**break;**

**// Hijo**

**case 0:**

**nsid = setsid();**

**if (nsid == -1) {**

**perror("Unable to create a new session");**

**exit(-1);**

**}**

**if ((chdir("/tmp")) == -1) {**

**perror("Unable to change the CWD");**

**exit(-1);**

**}**

**showProcessInfo("Child");**

**exit(0);**

**break;**

**// Padre**

**default:**

**showProcessInfo("Parent");**

**int status;**

**wait(&status);**

**if (status == -1) {**

**perror("Child process failed");**

**return -1;**

**}**

**}**

**return 0;**

**}**

Apartado 3: Ejecución de Programas

***Ejercicio 1.*** Las funciones principales para la ejecución de programas son system() y la familia de llamadas exec. Escribir dos programas, uno con system y otro con la llamada exec adecuada, que ejecute un programa que se pasará como argumento por línea de comandos. En cada caso, después de la ejecución mostrar la cadena “El comando terminó de ejecutarse”.

**Nota:** Considerar cómo deben pasarse los argumentos en cada caso para que sea sencilla la implementación. Ejemplo: ¿Qué diferencia hay entre **ejecuta ps -el** y **ejecuta “ps -el”** ?

Sin comillas dobles, le estás pasando dos argumentos a exec y con comillas solo uno. En cambio, en system al usar como manejador a la terminal siempre le pasas un argumento.

1a:

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

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

**//argv[1] = comando argv[2-n] = parametros**

**if(argc < 2) {**

**printf("Usage: command <argv1> <argv2> ...");**

**exit(EXIT\_FAILURE);**

**}**

**int i;**

**char command[255];**

**strcpy(command, argv[1]);**

**for(i= 2; i < argc; i++) {**

**strcat(command, " ");**

**strcat(command,argv[i]);**

**}**

**system(command);**

**return 0;**

**}**

1b:

**#include <unistd.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <errno.h>**

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

**if (argc < 2) {**

**printf("Usage: ./ej1b command [args]\n");**

**return -1;**

**}**

**int rc = execvp(argv[1], argv + 1);**

**if (rc == -1) {**

**perror("execvp()");**

**exit(-1);**

**}**

**printf("El comando termino de ejecutarse\n");**

**return 0;**

**}**

***Ejercicio 2.*** *Super-shell* . Escribir un programa que ejecute una shell (e.j. /bin/bash) con una política de planificación de tiempo real y la prioridad indicada como primer argumento. Si no se especifica debe fijar la prioridad máxima de la política:

● Arrancar una shell de prioridad máxima y ejecutar un bucle while infinito en un script bash (ej. bash -c ‘while true; do….’) con prioridad 12 y ayudate del comando chrt. Comprobar si es posible detener el proceso.

● Repetir el programa anterior pero usando una super-shell de prioridad 1.

**#include <sched.h>**

**#include <stdio.h>**

**#include <sys/types.h>**

**#include <unistd.h>**

**#include <errno.h>**

**#include <sys/time.h>**

**#include <sys/resource.h>**

**#include <stdlib.h>**

**#include <string.h>**

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

**int sch\_num = sched\_getscheduler(0);**

**if (sch\_num == -1) {**

**perror("Unable to get the scheduler");**

**return -1;**

**}**

**int prior\_min = sched\_get\_priority\_min(SCHED\_RR);**

**int prior\_max = sched\_get\_priority\_max(SCHED\_RR);**

**int prior = prior\_max;**

**if (argc == 2) {**

**int prior\_attr = atoi(argv[1]);**

**if (prior\_min <= prior\_attr && prior\_attr <= prior\_max)**

**prior = prior\_attr;**

**}**

**struct sched\_param prior\_param;**

**int rc\_prior = sched\_getparam(0, &prior\_param);**

**if (rc\_prior == -1) {**

**perror("Unable to get the scheduler priority");**

**return -1;**

**}**

**prior\_param.sched\_priority = prior;**

**int rc = sched\_setscheduler(0, SCHED\_RR, &prior\_param);**

**if (rc == -1) {**

**perror("Unable to set the scheduler");**

**return -1;**

**}**

**printf("Starting super-shell:\n");**

**system("bash -c 'while true; do echo HOLA; done'");**

**return 0;**

**}**

***Ejercicio 3.*** Usando la versión exec del ejercicio 1, y la plantilla de demonio desarrollada en la sección anterior, escribir un programa “demonize” que ejecute cualquier programa como si fuera un demonio.

Finalmente redirigir las salidas asociadas al terminal usando dup2, de forma que:

● stdout al fichero /tmp/daemon.out

● stderr al fichero /tmp/daemon.err

● stdin a /dev/null

Comprobar que al cerrar la shell el proceso sigue en ejecución.

**#include <sys/types.h>**

**#include <sys/stat.h>**

**#include <fcntl.h>**

**#include <unistd.h>**

**#include <stdlib.h>**

**#include <sys/time.h>**

**#include <sys/resource.h>**

**#include <stdio.h>**

**#include <errno.h>**

**#include <sys/types.h>**

**#include <sys/wait.h>**

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

**if (argc < 2) {**

**printf("Faltan argumentos, uso: ./ej3 program [args]\n");**

**return -1;**

**}**

**pid\_t pid = fork();**

**pid\_t nsid;**

**int status;**

**switch (pid) {**

**case -1:**

**perror("Unable to fork");**

**return -1;**

**break;**

**// Demonized child**

**case 0:**

**nsid = setsid();**

**if (nsid == -1) {**

**perror("Unable to create a new session");**

**exit(-1);**

**}**

**int ncwd = chdir("/tmp");**

**if (ncwd == -1) {**

**perror("Unable to change the CWD");**

**exit(-1);**

**}**

**int daemon\_in = open("/dev/null", O\_RDONLY);**

**int daemon\_out = open("daemon.out", O\_WRONLY | O\_CREAT);**

**int daemon\_err = open("daemon.err", O\_WRONLY | O\_CREAT);**

**if (daemon\_in == -1 || daemon\_out == -1 || daemon\_err == -1) {**

**perror("Failed to open in/out/err file:");**

**close(daemon\_in);**

**close(daemon\_out);**

**close(daemon\_err);**

**exit(-1);**

**}**

**int dup\_in = dup2(daemon\_in, 0);**

**if (dup\_in == -1) {**

**perror("Unable to duplicate the stdin:");**

**exit(-1);**

**}**

**int dup\_out = dup2(daemon\_out, 1);**

**if (dup\_out == -1) {**

**perror("Unable to duplicate the stdout:");**

**exit(-1);**

**}**

**int dup\_err = dup2(daemon\_err, 2);**

**if (dup\_err == -1) {**

**perror("Unable to duplicate the error stderr:");**

**exit(-1);**

**}**

**int rc = execvp(argv[1], argv + 1);**

**if (rc == -1) {**

**perror("System call failed");**

**exit(-1);**

**}**

**exit(0);**

**break;**

**default:**

**wait(&status);**

**if (status == -1) {**

**perror("Command failed");**

**return -1;**

**}**

**printf("El comando termino de ejecutarse\n");**

**return 0;**

**break;**

**}**

**}**

Apartado 4: Señales

***Ejercicio 1.*** El comando kill permite enviar señales a un proceso, grupo de procesos por su identificador (la variante pkill permite hacerlo por nombre de proceso). Estudiar la página de manual del comando y las señales que se pueden enviar a un proceso.

***Ejercicio 2.*** En un terminal arrancar un proceso de larga duración (ej. sleep 600). En otra terminal enviar diferentes señales al proceso, comprobar el comportamiento. Observar el código de salida de sleep. ¿Qué relación hay con la señal enviada?

***Ejercicio 3.*** Escribir un programa que bloquee las señales SIGINT y SIGTSTP. Después de bloquearlas el programa debe suspender su ejecución con la llamada sleep un número de segundos que se obtendrán de la variable de entorno SLEEP\_SECS.

Después de despertar de la llamada sleep, el proceso debe informar de si se recibió la señal SIGINT y/o SIGTSTP. En este último caso debe desbloquearla con lo que el proceso se detendrá y podrá ser reanudado en la shell, imprimir una cadena antes de finalizar el programa para comprobar este comportamiento.

**SLEEP\_SECS="20"**

**export SLEEP\_SECS**

**gcc -o ej3 ej3.c**

**./ej3**

**#include <signal.h>**

**#include <stdio.h>**

**#include <errno.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**int main() {**

**sigset\_t mascara;**

**sigemptyset(&mascara);**

**sigaddset(&mascara, SIGINT);**

**sigaddset(&mascara, SIGTSTP);**

**if (sigprocmask(SIG\_BLOCK, &mascara, NULL) == -1) {**

**perror("Unable to block the signals");**

**return -1;**

**}**

**char \*sleep\_secs = getenv("SLEEP\_SECS");**

**if (sleep\_secs == NULL) {**

**printf("Unable to read SLEEP\_SECS\n");**

**return -1;**

**}**

**int secs = atoi(sleep\_secs);**

**if (secs < 0)**

**secs = 0;**

**sleep(secs);**

**sigset\_t pendiente;**

**sigemptyset(&pendiente);**

**if (sigpending(&pendiente) == -1) {**

**perror("Unable to check the pending signals");**

**return -1;**

**}**

**switch (sigismember(&pendiente, SIGINT)) {**

**case 1:**

**printf("SIGINT recieved during sleep\n");**

**break;**

**case -1:**

**perror("Unable to check the pending signals");**

**return -1;**

**break;**

**}**

**switch (sigismember(&pendiente, SIGTSTP)) {**

**case 1:**

**printf("SIGTSTP recieved during sleep, unblocking signal...\n");**

**sigdelset(&mascara, SIGINT);**

**if (sigprocmask(SIG\_UNBLOCK, &mascara, NULL) == -1) {**

**perror("Unable to block the signals");**

**return -1;**

**}**

**break;**

**case -1:**

**perror("Unable to check the pending signals");**

**return -1;**

**break;**

**}**

**printf("End of the program\n");**

**return 0;**

**}**

***Ejercicio 4.*** Escribir un programa que instale un manejador sencillo para las señales SIGINT y SIGTSTP.

El manejador debe contar las veces que ha recibido la señal. El programa principal permanecerá en un bucle que se detendrá cuando se hayan recibido 10 señales. El número de señales de cada tipo se mostrará al finalizar el programa.

**#include <signal.h>**

**#include <stdio.h>**

**#include <errno.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**volatile int intCount = 0;**

**volatile int tstpCount = 0;**

**void manejador(int signo) {**

**if (signo == SIGINT)**

**intCount++;**

**if (signo == SIGTSTP)**

**tstpCount++;**

**}**

**int main() {**

**struct sigaction handler;**

**if (sigaction(SIGINT, NULL, &handler) == -1) {**

**perror("Unable to get SIGINT handler");**

**return -1;**

**}**

**handler.sa\_handler = manejador;**

**if (sigaction(SIGINT, &handler, NULL) == -1) {**

**perror("Unable to set SIGINT handler");**

**return -1;**

**}**

**if (sigaction(SIGTSTP, NULL, &handler) == -1) {**

**perror("Unable to get SIGTSTP handler");**

**return -1;**

**}**

**handler.sa\_handler = manejador;**

**if (sigaction(SIGTSTP, &handler, NULL) == -1) {**

**perror("Unable to set SIGTSTP handler");**

**return -1;**

**}**

**sigset\_t mascara;**

**sigemptyset(&mascara);**

**while (intCount + tstpCount < 10)**

**sigsuspend(&mascara);**

**printf("SIGINT captured: %i\n", intCount);**

**printf("SIGTSTP captured: %i\n", tstpCount);**

**return 0;**

**}**

***Ejercicio 5.*** Auto-destrucción. Escribir un programa que realice el borrado programado del propio ejecutable. El programa tendrá como argumento el número de segundos que esperará antes de borrar el fichero. El borrado del fichero se podrá detener si se recibe la señal SIGUSR1.

**Nota:** El programa principal no se puede suspender usando la función sleep(). Usar las funciones del sistema para borrar el fichero.

**#include <signal.h>**

**#include <stdio.h>**

**#include <errno.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**volatile int stop = 0;**

**void manejador(int signo) {**

**if (signo == SIGUSR1)**

**stop = 1;**

**}**

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

**if (argc != 2) {**

**printf("Use: program secs\n");**

**return -1;**

**}**

**sigset\_t mask;**

**sigemptyset(&mask);**

**sigaddset(&mask, SIGUSR1);**

**if (sigprocmask(SIG\_UNBLOCK, &mask, NULL)) {**

**perror("Unable to unblock SIGUSR1");**

**return -1;**

**}**

**struct sigaction handler;**

**if (sigaction(SIGUSR1, NULL, &handler) == -1) {**

**perror("Unable to get SIGUSR1 handler");**

**return -1;**

**}**

**handler.sa\_handler = manejador;**

**if (sigaction(SIGUSR1, &handler, NULL) == -1) {**

**perror("Unable to set SIGUSR1 handler");**

**return -1;**

**}**

**sleep(atoi(argv[1]));**

**if (stop == 1) {**

**printf("Auto-destruction stopped\n");**

**return 0;**

**}**

**if (unlink(argv[0]) == -1) {**

**perror("Unable to delete myself");**

**return -1;**

**}**

**else**

**printf("Deleted myself\n");**

**exit(0);**

**return 0;**

**}**