SYSTEM PROGRAMMING

From the book by STEWART WEISS

Chapter 07

Process Architecture and Control

Concepts Covered

- Signals (From Chapter 05)
- Process creation
- Process synchronization

- *nohup, pgrep, ps, psg*
- *sigaction, sigprocmask, kill, raise, atexit, fork, execve, exit, on_exit, wait, waitpid, waitid*

Signals

- Signals are software interrupts.
- They are a mechanism for handling asynchronous events, such as Ctrl-C at a terminal.
- Most applications need to handle signals.

Sources of Signals

- The terminal
- Hardware
- Software
- Processes

- The header file *<signal.h>* contains signal definitions.

Signal Types

Name	Value	Default	Event	Note	Category
SIGHUP	1	Exit	Hangup		Termination
SIGINT	2	Exit	Interrupt		Termination
SIGQUIT	3	Core	Quit		Termination
SIGILL	4	Core	Illegal Instruction		Program Error
SIGTRAP	5	Core	Trace or Breakpoint Trap		Program Error
SIGABRT	6	Core	Abort		Program Error
SIGEMT	7	Core	Emulation Trap		Program Error
SIGFPE	8	Core	Arithmetic Exception		Program Error
SIGKILL	9	Exit	Killed		Termination
SIGBUS	10	Core	Bus Error	1	Program Error
SIGSEGV	11	Core	Segmentation Fault		Program Error
SIGSYS	12	Core	Bad System Call	1	Program Error
SIGPIPE	13	Exit	Broken Pipe		Operation Error
SIGALRM	14	Exit	Alarm Clock		Alarm
SIGTERM	15	Exit	Terminated		Termination
SIGUSR1	16	Exit	User Signal 1	1	Miscellaneous

Sending Signals

- A process can send a signal to another process using:
- *int kill(int processid, int signal);*
- *kill(942, SIGTERM);*
- A process can also send a signal to itself using:
- *int raise(int signal);*
- which is equivalent to
- *kill(getpid(), signal);*

Signal Handling

- A process can choose to respond all signals differently except for SIGKILL and SIGSTOP.
- SIGKILL and SIGSTOP always terminate the process.
- To handle a signal, the programmer defines a function called a signal handler.
- The signal handler is executed when the signal is received.

The *sigaction()* call

- The *sigaction()* system call allows a process to register a signal handler and to specify how it will respond to multiple arriving signals.

- *#include <signal.h>*
- *int sigaction(int signum, const struct sigaction* act, struct sigaction* oldaction);*

- where
- signum is the value of the signal to be handled,
- act is a pointer to a *sigaction* structure that specifies the handler, masks, and flags for the signal
- oldact is a pointer to a structure to hold the currently active *sigaction* data.

The *sigaction* Structure

```
struct sigaction { // POSIX compliant, new-style handler
  // pointer to signal handler
  void (*sa_sigaction) (int, siginfo_t *, void *);
  sigset_t sa_mask; // additional signals to block
  // during handling of the signal
  int sa_flags; // flags that affect behavior
};
```

Example

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <bits/siginfo.h>
#include <stdio.h>
#include <stdlib.h>
```

```
void sig_handler(int signo, siginfo_t* info, void* context) {
  printf("Signal number: %d\n", info->si_signo);
  printf("Error number: %d\n", info->si_errno);
  printf("PID of sender: %d\n", info->si_pid);
  printf("UID of sender: %d\n", info->si_uid);
  exit(1);
}
```

Example

```
int main(int argc, char* argv[]) {
  struct sigaction the_action;
  the_action.sa_flags = SA_SIGINFO;
  the_action.sa_sigaction = sig_handler;
  sigaction(SIGINT, &the_action, NULL);
  printf("Type Ctrl-C wi thin the next minute or send signal 2.\n");
  sleep(60);
  return 0;
}
```

Blocking Signals Temporarily: *sigprocmask()*

- The *sigprocmask()* system call can block or unblock signals sent to a process.
- This is useful if you need to temporarily turn off all signals in a small section of code.
- It does not prevent the kernel from preempting the process and letting another process run on the CPU.
- It allows the process to complete some critical sequence of statements without any signal handlers running in the middle, and without being terminated in the middle.

Blocking Signals Temporarily: *sigprocmask()*

- The prototype is:
- *int sigprocmask(int how, const sigset_t *sigs, sigset_t *prev);*
- where how is one of SIG_BLOCK, SIG_UNBLOCK, or SIG_SETMASK.
- SIG_BLOCK will block the specified signal set.
- SIG_UNBLOCK allows the signals in the set to be unblocked.
- SIG_SETMASK is used to change the mask completely, i.e., assign a new mask to the procmask.

Processes

- A process is defined to be **a program in execution**.
- A program such as the bash can have many instances running on a machine
- Each individual instance is a separate and distinct process.
- Each and every instance is executing the same executable file.

Examining Processes on the Command Line

- *ps* gives list of running and zombie processes:

$ ps -f

UID PID PPID C STIME TTY TIME CMD

sweiss 2508 2507 0 12:09 pts/8 00:00:00 -bash

sweiss 3132 2508 0 12:22 pts/8 00:00:00 ps –f

- *pgrep* gives the process id of a command or program that is running:

$ pgrep bash

2508

3502

3621

Process Groups

- UNIX systems allow processes to be placed into groups.
- It is useful, for example:
- A signal can be sent to an entire process group rather than a single process.

Foreground and Background Processes

- Processes invoked from a shell command line are foreground processes
- They may be placed into the background by appending an '&' to the command line.
- A background process can run even after a logout, by using the *nohup* command, so it will ignore SIGHUP signals, as in:

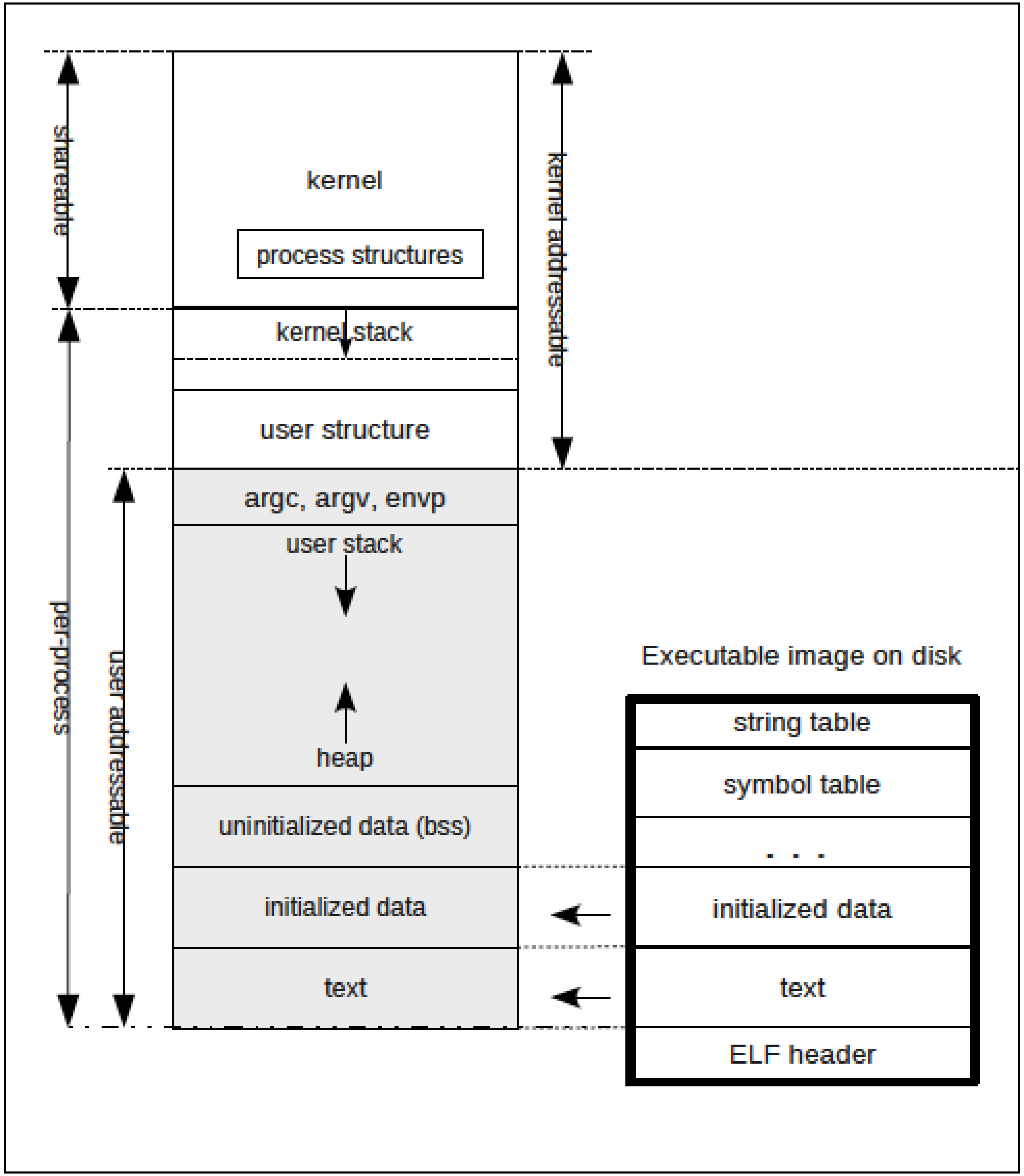
- *$ nohup do_backup &*

Sessions

- When a user logs on, the kernel;
  - Creates a session,
  - Places all processes and process groups of that user into the session,
  - Links the session to the terminal as its controlling terminal.

- Every session has a unique session-id of type pid_t.
- The primary purpose of sessions is to organize processes around their controlling terminals.

The Memory Architecture of a Process



Creating New Processes Using fork

- All processes are created with fork():

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

pid_t processid = fork();
```

causes the kernel to create a new process that is almost an exact copy of the calling process.

Creating New Processes Using fork

```
processid = fork();
if (processid == 0)
  // child's code here
else
  // parent's code here
```

Synchronizing Processes with Signals

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

void c_action(int signum) {
/*nothing to do here*/
}
```

```
int main(int argc, char* argv[]) {
  pid_t pid;
  int status;
  static struct sigaction childAct;

  switch (pid = fork()) {
    case - 1:
      perror ("fork() failed!");
      exit(1);
```

Synchronizing Processes with Signals

```
case 0: {
  /*child executes this branch, set SIGUSR1 action for child*/
  int i, x = 1;
  childAct.sa_handler = c_action;
  sigaction(SIGUSR1, &childAct, NULL);
  pause();
  printf("Child: starting computation... \n");
  for(i = 0; i < 10; i++) {
    printf("2^%d = %d\n", i, x);
    x = 2*x;
  }
  exit(0);
}
```

Synchronizing Processes with Signals

```
  default:
    /*parent code*/
    printf("Parent process: Will wait 2 seconds to prove child waits.\n");
    sleep(2); /*to prove that child waits for signal*/
    printf("Parent process: Sending child notice to start computation.\n");
    kill(pid, SIGUSR1);
    /*parent waits for child to return here*/
    if ((pid = wait(&status)) == -1) {
      perror("wait failed");
      exit(2);
    }
    printf("Parent process: child terminated.\n");
    exit(0);
  }
}
```

Executing Programs: The exec family

```
#include <unistd.h>
int execve(const char* filename, char* const argv[], char* const envp[]);
```

- execve() **executes the program** pointed to by its first argument.
- The filename must be a binary executable or a script whose first line is #! interpreter [optional-arg]

Executing Programs: The exec family

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

```
int main(int argc, char* argv[], char* envp []) {
  if (argc < 2) {
    printf("usage: execdemo1 arg1 [arg2 ...]\n");
    exit(1);
  }

  execve("/bin/echo", argv, envp);
  fprintf(stderr, "execve() failed to run.\n");
  exit(1);
}
```

The fprintf() statement will only be executed if the execve() call fails; execve() returns only when it fail to run.

Synchronizing Parents and Children: *wait* and *exit*

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

- Three actions take place when *exit()* is called:

1. The process's registered exit functions run;
2. The system gets a chance to clean up after the process;
3. The process gets a chance to have a status value delivered to its parent.

Registering *exit* Functions

- Programmers can register a function to run when a process calls *exit()* using either *atexit()* or *on_exit()*.

Waiting for Children to Terminate

- After a process forks a child, how will it know if and when the child has finished its task?
- A process has to wait until the child or children finish their tasks before it can continue.

The *wait()* family of calls

- There are three different POSIX-compliant *wait()* system calls

```
#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);
```

- These system calls;
- delay the parent until a child has terminated,
- obtain the status of a child that has terminated.

Example for *wait()*

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
```

```
void child() {
  int exit_code;
  printf("I am the child. My pid: %d.\n", getpid());
  sleep(2);
  printf("Enter a value for the child exit code:\n");
  scanf("%d", &exit_code);
  exit(exit_code);
}
```

Example for *wait()*

```
int main(int argc, char* argv[]) {
  int pid, status;
  printf("Starting up... \n");
  if (-1 == (pid = fork())) {
    perror("fork"); exit(1);
  }
  else if (0 == pid)
    child();
  else { /*parent code*/
    printf("My child has pid %d and my pid is %d.\n", pid, getpid());
    if ((pid = wait(&status)) == -1) {
      perror("wait failed"); exit(2);
    }
```

Example for *wait()*

```
    if (WIFEXITED(status)) { /*low order byte of status equals 0 */
      printf("Parent: Child %d exited with status %d.\n",
              pid, WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
      printf("Parent: Child %d exited with err. code %d.\n",
              pid, WTERMSIG(status));
#ifdef WCOREDUMP
      if (WCOREDUMP(status))
        printf("Parent: A core dump took place.\n");
#endif
    }
  }
  return 0;
}
```

Using *waitpid()*

- The *waitpid()* function has three parameters:

- The process-id of the child to wait for,
- A pointer to the variable in which to store the status,
- An optional set of flags.

Non-blocking waits

- Instead of calling *wait()* or *waitpid()*, a process can establish a SIGCHLD handler that will run when a child terminates.
- The SIGCHLD handler can then call *wait()*.

- This frees the process from having to poll the *wait()* function.
- It only calls *wait()* when it is guaranteed to succeed.

- Check Listing 7.13 for example code!

Thanks...