6. Process Synchronization and Event Flow

Outline

- Motivating example
- Process synchronization
  - Monitoring process state change (termination)
  - Delivering and catching signals
- Programmer interface
  - Main system calls
  - Examples

Motivating Example

Shell Job Control

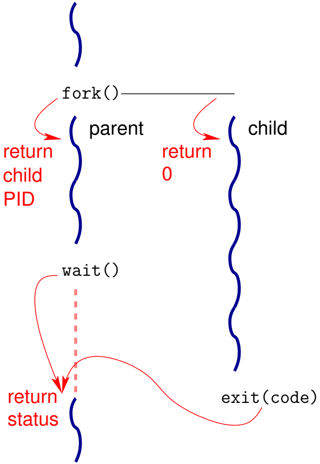
Monitoring stop/resume cycles of a child process

```
$ sleep 60
Ctrl-Z                     // Deliver SIGTSTP

[1]+  Stopped       sleep  // Recieved terminal stop signal
$ kill -CONT %1            // Equivalent to fg
sleep                      // Resume process
Ctrl-C                     // Deliver SIGINT
                           // Terminate process calling exit(0)
$
```

How does this work?

Monitoring Processes



System Call: wait() and waitpid()

Wait For Child Process to Change State

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

Description

- Monitor state changes and return PID of
  - Terminated child
  - Child stopped by a signal
  - Child resumed by a signal
- If a child terminates, it remains in a *zombie* state until `wait()` is performed to retrieve its state (and free the associated process descriptor)
  - Zombie processes do not have children: they are adopted upon termination by `init` process (**1**)
  - The `init` process always waits for its children
  - Hence, a zombie is removed when its parent terminates

System Call: wait() and waitpid()

Wait For Child Process to Change State

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

Whom to Wait For

- pid > 0 : waitpid() suspends process execution until child specified by pid changes state, or returns immediately if it already did
- pid $= 0$: wait for any child in the same process group
- pid < -1: wait for any child in process group -pid
- pid $= -1$: wait for any child process

Short Cut

wait(&status) is equivalent to waitpid(-1, &status, 0)

System Call: wait() and waitpid()

Wait For Child Process to Change State

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

How to Wait

- Option WNOHANG: do not block if no child changed state
  Return **0** in this case
- Option WUNTRACED: report stopped child
  (due to SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU signals)
- Option WCONTINUED: report resumed child
  (due to SIGCONT signal)

System Call: wait() and waitpid()

Wait For Child Process to Change State

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

State Change Status

- If non-NULL status_pointer, store information into the int it points to

  WIFEXITED(status): true if child terminated normally (i.e., _exit())
  WEXITSTATUS(status): if the former is true, child exit status
  (lower **8** bits of status)
  WIFSIGNALED(status): true if child terminated by signal
  WTERMSIG(status): if the former is true, signal that caused termination
  WIFSTOPPED(status): true if child stopped by signal
  WSTOPSIG(status): if the former is true, signal that caused it to stop
  WIFCONTINUED(status): true if child was resumed by delivery of SIGCONT

System Call: wait() and waitpid()

Wait For Child Process to Change State

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

Error Conditions

- Return $-\mathbf{1}$ if an error occurred
- Typical error code

  ECHILD, calling wait(): if all children were configured to be *unattended* (a.k.a. *un-waited for*, i.e., not becoming zombie when terminating, *see* sigaction())

  ECHILD, calling waitpid(): pid is not a child or is *unattended*

Process State Changes and Signals

Process State Monitoring Example

```c
int main(int argc, char *argv[])
{
  int status;
  cpid = fork();
  if (cpid == -1) { perror("fork"); exit(1); }
  if (cpid == 0) {                 // Code executed by child
    printf("Child PID is %ld\n", (long)getpid());
    pause();                       // Wait for signals
  } else {                         // Code executed by parent
    do {
      pid_t w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
      if (w == -1) { perror("waitpid"); exit(1); }
      if (WIFEXITED(status)) {     // Control never reaches this point
        printf("exited, status=%d\n", WEXITSTATUS(status));
      } else if (WIFSIGNALED(status)) {
        printf("killed by signal %d\n", WTERMSIG(status));
      } else if (WIFSTOPPED(status)) {
        printf("stopped by signal %d\n", WSTOPSIG(status));
      } else if (WIFCONTINUED(status)) { printf("continued\n"); }
    } while (!WIFEXITED(status) && !WIFSIGNALED(status));
  }
  exit(0);
}
```

Process State Changes and Signals

Running the Process State Monitoring Example

```
$ ./a.out &
Child PID is 32360
[1] 32359
$ kill -STOP 32360
stopped by signal 19
$ kill -CONT 32360
continued
$ kill -TERM 32360
killed by signal 15
[1]+  Done                    ./a.out
$
```

Process Synchronization With Signals

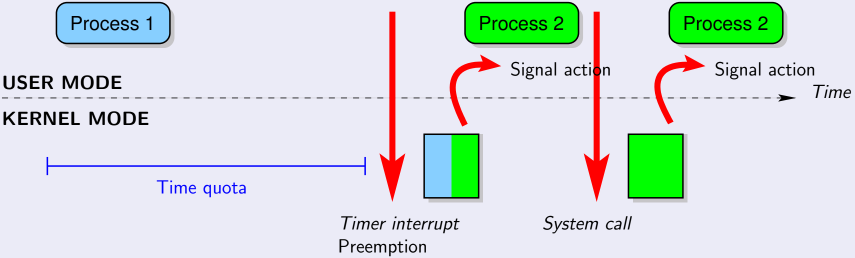
Principles

- Signal *delivery* is *asynchronous*
  - Both sending and recieving are asynchronous
  - Sending may occur during the signaled process execution or not
  - Recieving a signal may interrupt process execution at an arbitrary point
- A signal *handler* may be called upon signal delivery
  - It runs in *user mode* (sharing the user mode stack)
  - It is called "*catching the signal*"
- A signal is *pending* if it has been delivered but not yet handled, because it is currently *blocked*
  (or because the kernel did not yet check for its delivery status)
- *No queueing* of pending signals

Process Synchronization With Signals

Catching Signals

- Signal caught when the process *switches from kernel to user mode*
  - Upon context switch
  - Upon return from system call



System Call: kill()

Send a Signal to a Process or Probe for a Process

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Whom to Deliver the Signal

pid > 0 : to pid

pid $= 0$: to all processes in the group of the current process

pid < -1: to all processes in group $-$pid

pid $= -1$: to all processes the current process has permitssion to send signals to, except himself and init (1)

System Call: kill()

Send a Signal to a Process or Probe for a Process

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Existence and Permission

- No signal sent if sig is **0**, but error checks are performed
- The real or (saved) effective UID of the sender must match the real or (saved) effective UID of the reciever

Error Conditions

- Return **0** on success, **−1** on error
- Possible errno codes

  EINVAL: an invalid signal was specified

  EPERM: no permission to send signal to any of the target processes

  ESRCH: the process or process group does not exist

List of The Main Signals

SIGHUP[0]: terminal hang up

SIGINT[0]: keyboard interrupt (**Ctrl-C**)

SIGQUIT[0,1]: keyboard quit (**Ctrl**-\)

SIGKILL[0,3]: unblockable kill signal, terminate the process

SIGBUS/SIGSEGV[0,1]: memory bus error / segmentation violation

SIGSYS[0]: bad system call or argument

SIGPIPE[0]: broken pipe (writing to a pipe with no reader)

SIGALRM[0]: alarm signal

SIGTERM[0]: termination signal (`kill` command default)

SIGSTOP[3,4]: suspend process execution,

SIGTSTP[4]: terminal suspend (**Ctrl-Z**)

SIGTTIN/SIGTTOU[4]: terminal input/output for background process

SIGCONT[2]: resume after (any) suspend

SIGCHLD[2]: child stopped or terminated

SIGUSR1[0]: user defined signal 1

SIGUSR2[0]: user defined signal 2

More signals: `$ man 7 signal`

[0] terminate process

[1] dump a core

[2] ignored by default

[3] non-maskable, non-catchable

[4] suspend process

System Call: signal()

ISO C Signal Handling (pseudo UNIX V7

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);

// Alternate, "all-in-one" prototype
void (*signal(int signum, void (*handler)(int)))(int);
```

Description

- Install a new handler for signal `signum`
  - `SIG_DFL`: default action
  - `SIG_IGN`: signal is ignored
  - Custom handler: function pointer of type `sighandler_t`
- Return the previous handler or `SIG_ERR`
- Warning: unsupported in multi-threaded or real-time code (linked with `-lrt`)
  The Forum application used in labs is threaded and linked with the real-time library

System Call: signal()

ISO C Signal Handling (pseudo UNIX V7

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);

// Alternate, "all-in-one" prototype
void (*signal(int signum, void (*handler)(int)))(int);
```

When Executing the Signal Handler

- The `signum` argument is the caught signal number
- Blocks (defers) nested delivery of the signal being caught
- On UNIX V7, it used to reinstall the default handler instead
- Asynchronous execution w.r.t. the process's main program flow
  - Careful access to global variables (much like threads)
  - Limited opportunities for system calls
    Explicit list of "safe" functions: `$ man 2 signal`

System Call: `pause()`

Wait For Signal

```
#include <unistd.h>

int pause();
```

Description

- Suspends the process until it is delivered a signal
  - That terminate the process (`pause()` does not return...)
  - That causes a signal handler to be called
- Ignored signals (`SIG_IGN`) do *not* resume execution
  In fact, they never interrupt any system call
- Always return $-\mathbf{1}$ with error code `EINTR`

System Call: alarm()

Set an Alarm Clock for Delivery of a SIGALRM

```
#include <unistd.h>

int alarm(unsigned int seconds);
```

Description

- Deliver SIGALRM to the calling process after a delay (non-guaranteed to react immediately)
- Warning: the default action is to terminate the process

System Call: alarm()

C library function: sleep

```
unsigned int sleep(unsigned int seconds)
```

- Combines signal(), alarm() and pause()
- Uses the same timer as alarm() (hence, do not mix)
- See also setitimer()

Putting the Process to Sleep

```
void do_nothing(int signum)
{
  return;
}

void my_sleep(unsigned int seconds)
{
  signal(SIGALRM, do_nothing); // Note: SIG_IGN would block for ever!
  alarm(seconds);
  pause();
  signal(SIGALRM, SIG_DFL);    // Restore default action
}
```

More Complex Event Flow Example

Shell Job Control

Monitoring stop/resume cycles of a child process

```
$ top
Ctrl-Z                     // Deliver SIGTSTP

[1]+  Stopped         top  // Stop process
$ kill -CONT %1            // Resume (equivalent to fg)
                           // Recieve SIGTTOU and stop
[1]+  Stopped         top  // Because of background terminal I/O
$ kill -INT %1
                           // SIGINT is pending, i.e.
[1]+  Stopped         top  //   did not trigger an action yet
$ fg
top
                           // Terminate process calling exit(0)
$
```