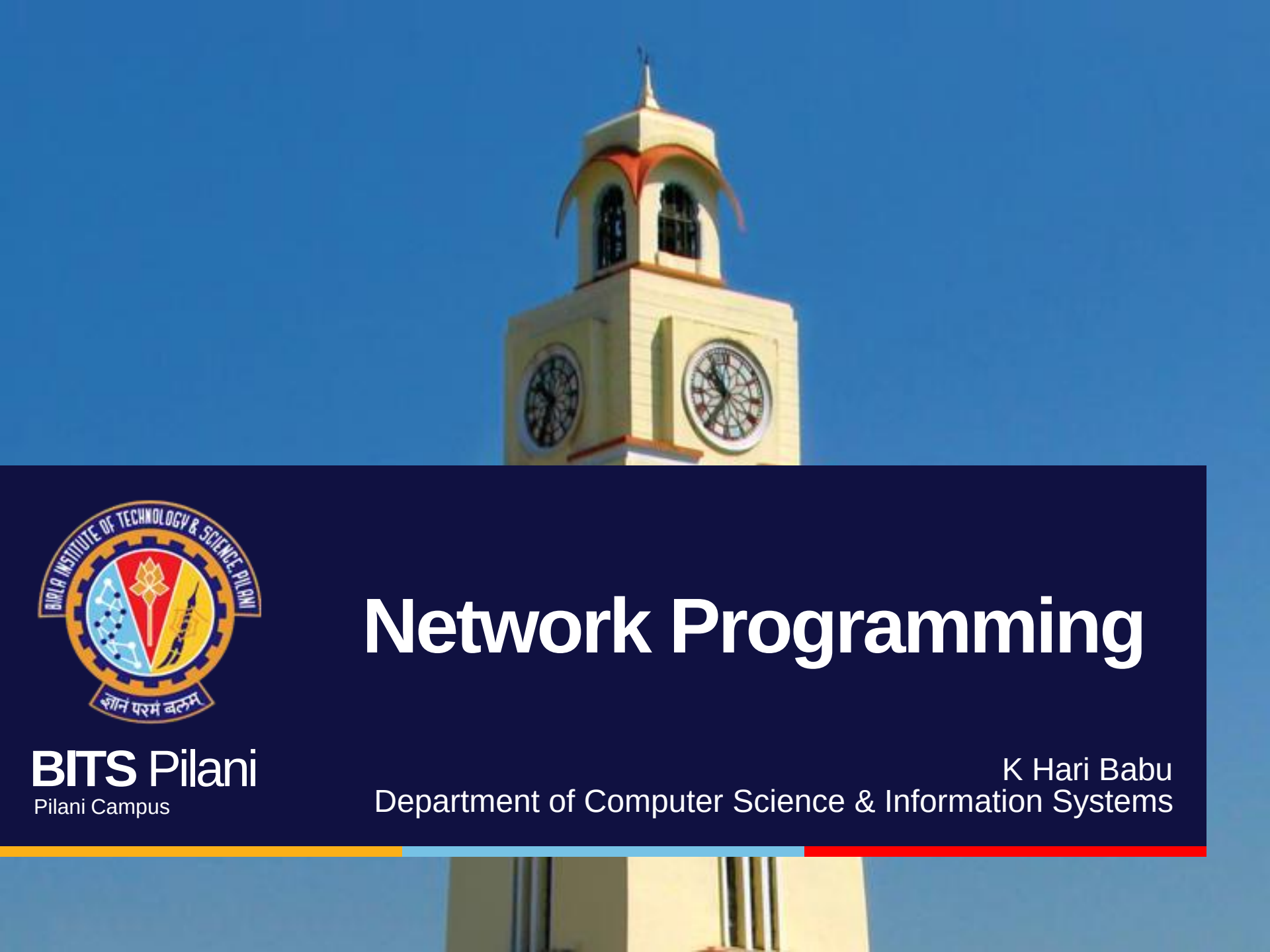


Network Programming

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

Outline

- Process creation
  - *fork()*
  - File sharing between parent and child
  - Copy-on-write
  - *vfork()*
- Process termination
  - *exit() & _exit()*
- Monitoring child processes
  - *wait() & waitpid()*
  - Wait status
  - Orphans & zombies
- Program execution
  - *execve()*
  - File descriptors

Outline

- Signals
  - Definition
  - Signal disposition
  - Signal handler
  - Signal generation
  - Signal mask
  - Waiting for a signal
    - Asynchronously
    - Synchronously

innovate achieve lead

BITS Pilani

Pilani Campus

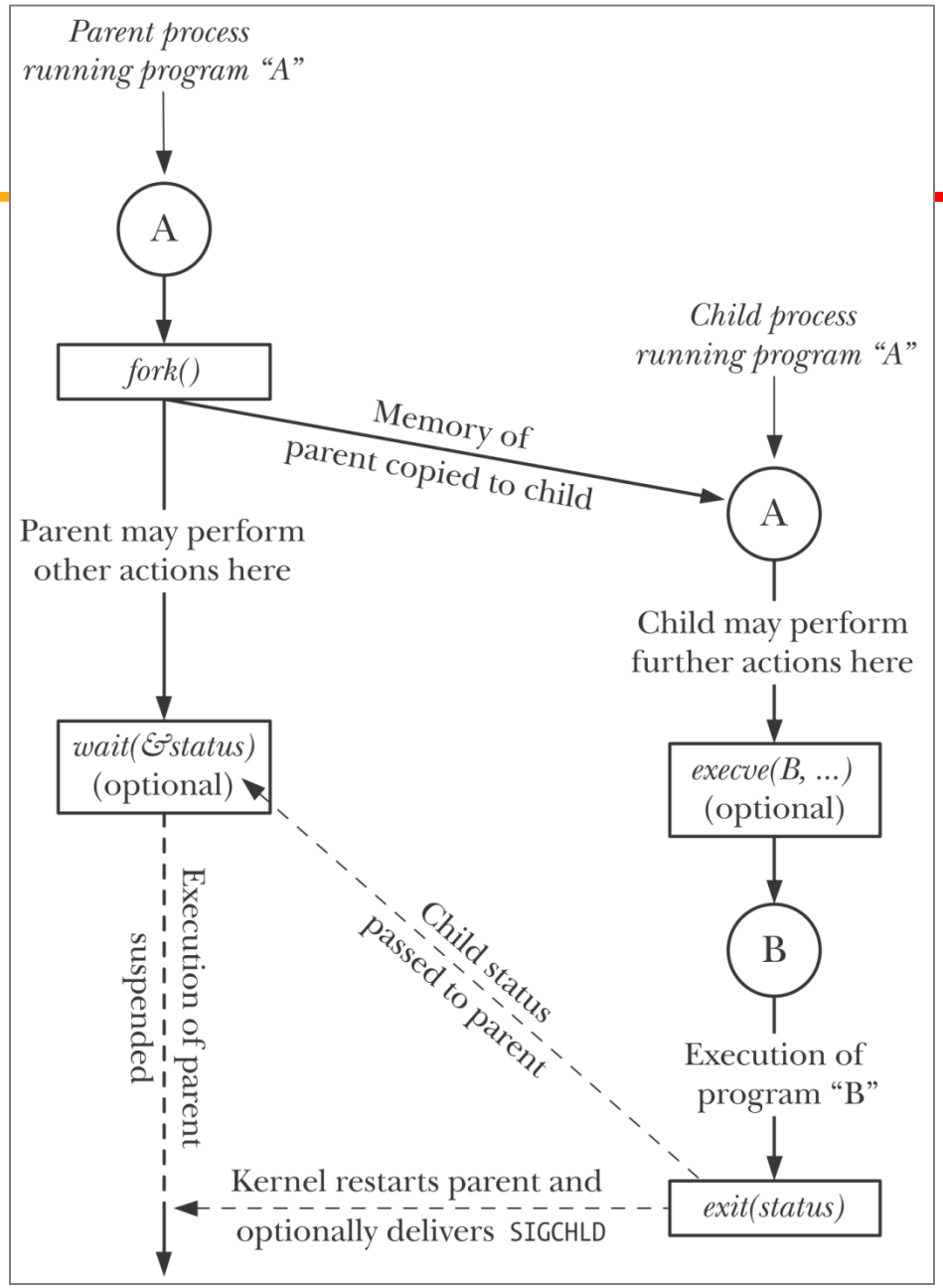
Process Creation (R1: Ch24)

Process Creation

- An existing process can create a new process by calling fork function. The new process created by fork is called *child process*.

```
2  #include <unistd.h>
3  pid_t fork(void);
4  /*In parent: returns process ID of child on success, or -1 on error;
5  in successfully created child: always returns 0*/
```

  - This function is called once but returns twice.
    - The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
  - The child is a copy of the parent. The child gets a copy of the parent's data section, heap, and the stack. Memory is copied not shared.
  - The parent and the child share the text segment.


Parent process
running program "A"
A
fork()
Memory of
parent copied to child
Child process
running program "A"
A
Parent may perform
other actions here
Child may perform
further actions here
wait(&status)
(optional)
execve(B, ...)
(optional)
Child status
passed to parent
Execution of parent
suspended
B
Execution of
program "B"
Kernel restarts parent and
optionally delivers SIGCHLD
exit(status)

PIDs


```
2   pid_t childPid;             /* Used in parent after successful fork()
3                                  to record PID of child */
4   switch (childPid = fork()) {
5   case -1:                    /* fork() failed */
6       /* Handle error */
7   case 0:                     /* Child of successful fork() comes here */
8       /* Perform actions specific to child */
9   default:                    /* Parent comes here after successful fork() */
10      /* Perform actions specific to parent */
11  }
```

- Within the code of the program, child and parent can be distinguished by the return value of *fork()*.
  - In parent return value>0
  - In child return value==0
- In general, we never know whether the child starts executing before the parent or vice versa.
- To synchronize child and parent, some form of interprocess communication is required.

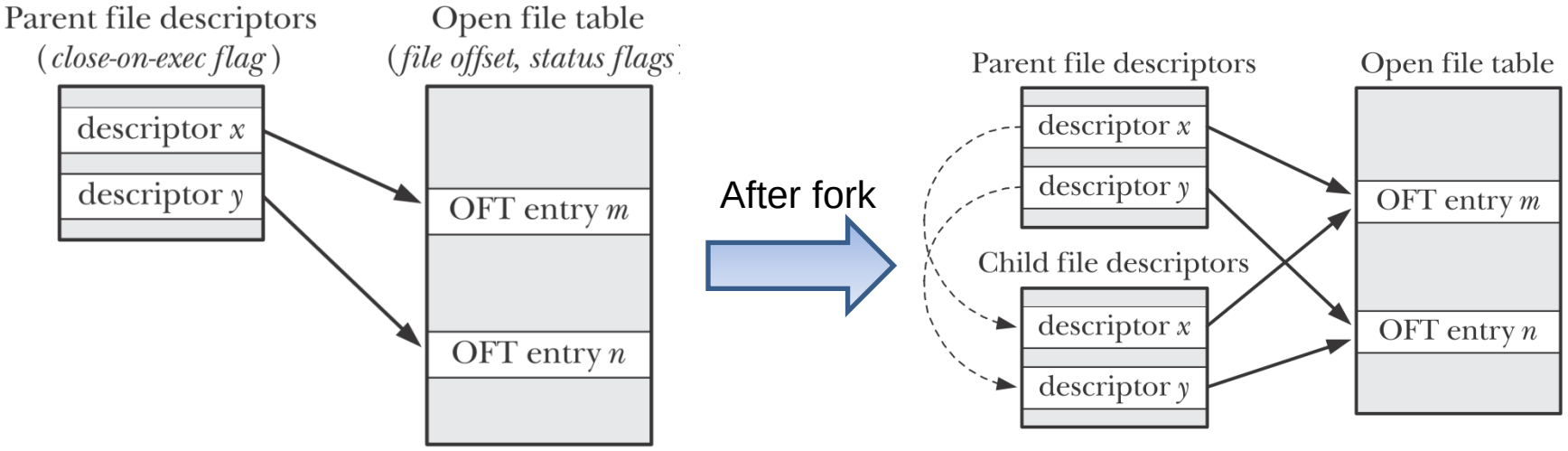
fork() demo


```
2   #include "tlpi_hdr.h"
3   static int idata = 111;                 /* Allocated in data segment */
4   int
5   main(int argc, char *argv[])
6   {
7       int istack = 222;                   /* Allocated in stack segment */
8       pid_t childPid;
9       switch (childPid = fork()) {
10      case -1:
11          errExit("fork");
12      case 0:
13          idata *= 3;
14          istack *= 3;
15          break;
16      default:
17          sleep(3);                       /* Give child a chance to execute */
18          break;
19      }
20      /* Both parent and child come here */
21      printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
22              (childPid == 0) ? "(child) " : "(parent)", idata, istack);
23      exit(EXIT_SUCCESS);
24  }
```

```
2   $ ./t_fork
3   PID=28557 (child)  idata=333 istack=666
4   PID=28556 (parent) idata=111 istack=222
```

File Sharing Between Parent and Child

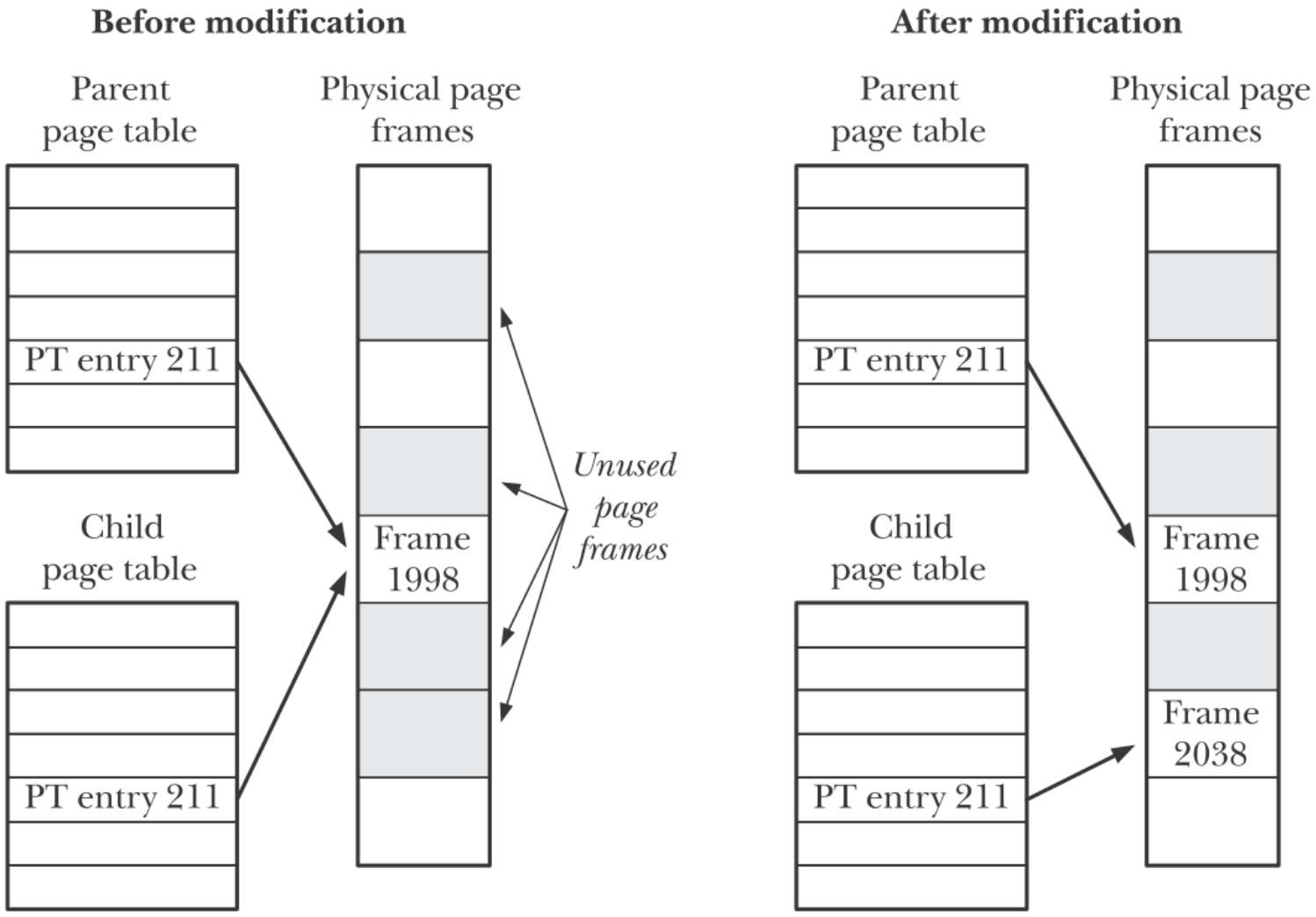
- When a *fork()* is performed, the child receives duplicates of all of the parent's file descriptors.
- Open file attributes are shared between the parent and the child.
  - Sharing the file offset:
    - parent and child do not over write each other's output.
    - but the outputs may be randomly intermingled. IPC required.



fork(): Copy-on-write

- Old UNIX implementations used to literally make copy of the parents memory.
  - This is wasteful.
- Modern UNIX implementations use two techniques to avoid it.
  - Mark text segment as read-only. Build child's page table such away that pages in text segment point to the same physical memory pages as those of parent.
  - Data, Heap, Stack segments: *copy on write*.
    - Mark these segments as read-only. Build child's page tables such a way that they point to the physical pages used by parent.
    - Kernel traps any attempt to modify the page. Makes a duplicate of the page and modifies the page table of the modifying process to point to this page.

Copy on Write: Page Tables



Process's Memory Footprint

- Process's memory footprint is the range of virtual memory pages used by the process.
- Controlling process's memory footprint:
  - Executing functions known for memory leaks. Execute such functions within a child process. That won't affect the parent.

```
2  /*from procexec/footprint.c*/
3  pid_t childPid;
4  int status;
5  childPid = fork();
6  if (childPid == -1)
7      errExit("fork");
8  if (childPid == 0)                  /* Child calls func() and */
9      exit(func(arg));                /* uses return value as exit status */
10 /* Parent waits for child to terminate. It can determine the
11    result of func() by inspecting 'status'. */
12 if (wait(&status) == -1)
13     errExit("wait");
```

vfork() system call

- In early UNIX implementations, *vfork()* was proposed as efficient version of fork()system call.
  - With the copy-on-write implementation, eliminated need for *vfork()*.
- *vfork()* is designed to be used where the child performs an immediate *exec()* call.
- Differences between vfork() and fork()
  - No duplication of page tables for the child.
    - Child shares parent's memory until it calls *_exit()* or *exec()*
    - Changes made by the child will be visible to parent.
  - Execution of the parent is suspended until the child has performed an *exec() or _exit().*

```
2  #include <unistd.h>
3  pid_t vfork(void);
4  /*In parent: returns process ID of child on success, or -1 on error;
5  in successfully created child: always returns 0*/
```

vfork() demo

innovate achieve lead

```c
/*procexec/t_vfork.c*/
#include "tlpi_hdr.h"
int
main(int argc, char *argv[])
{
    int istack = 222;
    switch (vfork()) {
    case -1:
        errExit("vfork");
    case 0:             /* Child executes first, in parent's memory space */
        sleep(3);                   /* Even if we sleep for a while,
                                       parent still is not scheduled */
        write(STDOUT_FILENO, "Child executing\n", 16);
        istack *= 3;                /* This change will be seen by parent */
        _exit(EXIT_SUCCESS);
    default:            /* Parent is blocked until child exits */
        write(STDOUT_FILENO, "Parent executing\n", 17);
        printf("istack=%d\n", istack);
        exit(EXIT_SUCCESS);
    }
}
```

```
$ ./t_vfork
Child executing /*Even though child slept, parent was not scheduled*/
Parent executing
istack=666
```

Race Conditions

- After fork(), it is indeterminate which process parent or child next has access to the CPU.
- Assuming a particular order of execution (e.g. parent and child) can lead to race conditions.
  - Race condition means that the result of execution will vary depending on the order of execution.
- A particular order can be achieved using either of the following
  - Signals
  - Pipes/message queues
  - Semaphores etc.

innovate achieve lead

BITS Pilani
Pilani Campus

Process Termination (R1: Ch25)

Process Termination

- A process may terminate in two ways
  - Abnormal termination
    - Cause by a signal such as SIGSEGV.
  - Normal termination
    - By calling _exit()

```
2  #include <unistd.h>
3  void _exit(int  status );
4  /*this function never returns*/
```

  - *_exit()* is a system call. *exit()* is a library call which invokes *_exit()*.
  - status is made available to the parent by the kernel. Only lower 8 bits of the *status* variable.
  - Generally status=0 means successfully completed.

Process Termination

- Regardless of how a process terminates, the same code in the kernel is eventually executed.
- During both normal and abnormal termination of a process
  - Open file descriptors are closed and file locks (if any) are released.
  - Shared memory segments or memory mappings are released.
  - semadj value is adjusted.
  - Kernel releases the memory that it was using, and the like.
- When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent.
- Kernel stores the exit status, resource usage statistics for the parent to know.

exit() – Library Function


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

- During execution of exit()
  - Exit handlers are called.
  - *stdio* stream buffers are flushed.
  - _exit() system call is invoked.
- Exit handlers:

```
2   #include <stdlib.h>
3   int atexit(void (* func )(void));
4       /*Returns 0 on success, or nonzero on error*/
```

  - Exit handlers are useful for cleanup tasks at the time of termination.
  - *atexit()* adds a *func* to the list of functions to be called at the time of termination.
  - Functions are invoked in reverse order.
  - Child inherits the exit handlers of the parent.

innovate achieve lead

BITS Pilani

Pilani Campus

Monitoring Child Processes (R1: Ch26)

Monitoring Child Processes

- It is useful for the parent process to know when and how a child process got terminated
- wait() or waitpid() system call is used to know the status of the child.
- wait() system call:

```
2  #include <sys/wait.h>
3  pid_t wait(int * status );
4  /*Returns process ID of terminated child, or -1 on error*/
```

  - Waits for one of the children to terminate and returns the termination status in *status* buffer.
  - It blocks until one of the child terminates.
  - It returns the pid of the child process.
  - returns -1 in case there are no more children.

waitpid() System Call

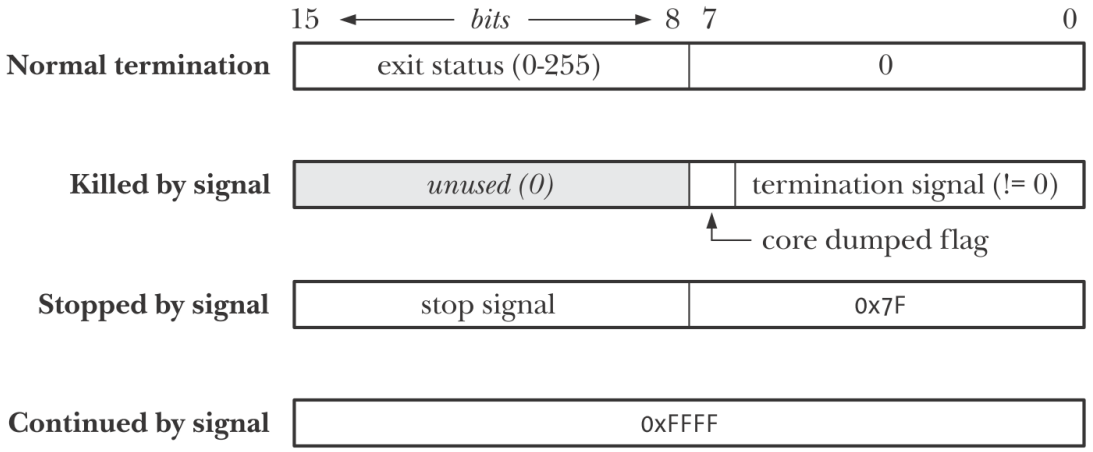
- wait() system call has limitations
  - Can't wait for a specific child
  - It always blocks until some child terminates.
  - Can't find out about children which are stopped or continued by a signal.
- waitpid() system call:

```
2   #include <sys/wait.h>
3   pid_t waitpid(pid_t  pid , int * status , int  options );
4   /*Returns process ID of child, 0 (WNOHANG option), or -1 on error*/
```

  - If pid>0, wait for a specific child process. If pid=-1, then same as wait().
  - Options:
    - WUNTRACED: child stopped by a signal SIGSTOP or SIGTTIN
    - WCONTINUED: child continued by SIGCONT signal
    - WNOHANG: do not block if the child is not terminated yet. Return 0.

Wait Status Value

- Although int (4 bytes) but only lower 2 bytes are used.
- There are macros to extract this info from the *status* returned by wait.




```
2   if (WIFEXITED (status))
3       printf ("normal termination, exit status = %d\n", WEXITSTATUS (status));
4   else if (WIFSIGNALED (status))
5     printf ("abnormal termination, signal number = %d \n", WTERMSIG (status));
6   else if (WIFSTOPPED (status))
7       printf ("child stopped, signal number = %d\n", WSTOPSIG (status));
8   else if (WIFCONTINUED (status)) /*available since Linux 2.6.10*/
9       printf ("child continued");
10  }
```

Orphan & Zombie Process

innovate achieve lead

- Orphan process: What happens if parent terminates before child?
  - the init process becomes the parent process of any process whose parent terminates (process has been inherited by init)
  - parent process ID of the surviving process is changed to be 1 (the process ID of init). This way, we're guaranteed that every process has a parent.
- Zombie process: What happens when a child terminates before its parent ?
  - Kernel keeps information (process ID, the termination status of the process, and the amount of CPU time taken by the process) until parent asks for it.
  - a process that has terminated, but whose parent has not yet waited for it, is called a zombie.
  - Zombies can hold up pids and resources in long-live servers.

innovate achieve lead

BITS Pilani
Pilani Campus

Program Execution (R1: Ch27)

Executing a New Program: *execve()*

- Replaces the program running in the process by a new program.
  - Process's text, stack, heap, data are replaced by those of the new program.
  - The process ID does not change across an exec, because a new process is not created;

```
2  #include <unistd.h>
3  int execve(const char * pathname , char *const  argv [], char *const  envp []);
4      /*Never returns on success; returns -1 on error*/
```

  - *pathname: absolute or relative path name of the image*
  - *Argv:* command-line arguments to be passed to the new program
  - *envp:* specifies the environment list for the new program.

Library Functions

innovate achieve lead

```
2   #include <unistd.h>
3   int execle(const char * pathname , const char * arg , ...
4                   /* , (char *) NULL, char *const  envp [] */ );
5   int execlp(const char * filename , const char * arg , ...
6                   /* , (char *) NULL */);
7   int execvp(const char * filename , char *const  argv []);
8   int execv(const char * pathname , char *const  argv []);
9   int execl(const char * pathname , const char * arg , ...
10                  /* , (char *) NULL */);
11  /*None of the above returns on success; all return -1 on error*/
```

- Above functions are library functions. They invoke execve() system call.
  - *l* = list of arguments
  - *v* = vector of arguments
  - *e* = environment specified.
  - *p* = look for file in $PATH environment variable.

Passing Arguments as a List


```c
/*procexec/t_execl.c*/
#include <stdlib.h>
#include "tlpi_hdr.h"
int
main(int argc, char *argv[])
{
    printf("Initial value of USER: %s\n", getenv("USER"));
    if (putenv("USER=britta") != 0)
        errExit("putenv");
    execl("/usr/bin/printenv", "printenv", "USER", "SHELL", (char *) NULL);
    errExit("execl");           /* If we get here, something went wrong */
}
```

- This is useful when we know the number of arguments at the time of writing program.

```
$ echo $USER $SHELL          Display some of the shell's environment variables
blv /bin/bash
$ ./t_execl
Initial value of USER: blv   Copy of environment was inherited from the shell
britta                       These two lines are displayed by execed printenv
/bin/bash
```

Passing Environment to New Program

innovate achieve lead

```
2  /*procexec/t_execle.c*/
3  #include "tlpi_hdr.h"
4  int
5  main(int argc, char *argv[])
6  {
7      char *envVec[] = { "GREET=salut", "BYE=adieu", NULL };
8      char *filename;
9      if (argc != 2 || strcmp(argv[1], "--help") == 0)
10         usageErr("%s pathname\n", argv[0]);
11     filename = strrchr(argv[1], '/');       /* Get basename from argv[1] */
12     if (filename != NULL)
13         filename++;
14     else
15         filename = argv[1];
16     execle(argv[1], filename, "hello world", (char *) NULL, envVec);
17     errExit("execle");           /* If we get here, something went wrong */
18 }
```

- Only those variables mentioned in *envVec* will be passed to new program.

File Descriptors and *exec()*

- By default all open file descriptors remain open across the exec().
  - The fds are available for use in the new program.
  - Shell takes advantage of this feature. e.g. > or |
- close-on-exec flag (FD_CLOEXEC)
  - FD_CLOEXEC is the only bit used in file descriptor's flags.
  - If this flag is set, then that fd will be closed upon successful exec().

```
2   int flags;
3   flags = fcntl(fd, F_GETFD);
4   if (flags == -1)
5       errExit("fcntl");
6   flags |= FD_CLOEXEC;
7   if (fcntl(fd, F_SETFD, flags) == -1)
8       errExit("fcntl");
```

innovate achieve lead

BITS Pilani

Pilani Campus

Signals (R1: Ch20)

What is a Signal?

innovate achieve lead

- A signal is a notification to a process that an event has occurred.
  - A signal is an *asynchronous* event which is delivered to a process.
  - Asynchronous means that the event can occur at any time
    - e.g. user types `ctrl-C`
- Source of signals:
  - Hardware exceptions
    - Dividing by 0, accessing inaccessible memory
  - User typing special characters at the terminal
    - Control-c, Control-\, Control-Z
  - Software events
    - Data available on a descriptor
    - A timer went off
    - Child is terminated etc.
  - kill function allows a process to send a signal to another process.

Signals

- Each signal is defined by a unique integer, starting from 1 to 31. 0 is a NULL signal.
  - Mapping varies across architectures. Better go by symbols.

Name	Signal number	Description	SUSv3	Default
SIGABRT	6	Abort process	•	core
SIGALRM	14	Real-time timer expired	•	term
SIGBUS	7 (SAMP=10)	Memory access error	•	core
SIGCHLD	17 (SA=20, MP=18)	Child terminated or stopped	•	ignore
SIGCONT	18 (SA=19, M=25, P=26)	Continue if stopped	•	cont
SIGEMT	undef (SAMP=7)	Hardware fault		term
SIGFPE	8	Arithmetic exception	•	core
SIGHUP	1	Hangup	•	term
SIGILL	4	Illegal instruction	•	core
SIGINT	2	Terminal interrupt	•	term
SIGIO / SIGPOLL	29 (SA=23, MP=22)	I/O possible	•	term
SIGKILL	9	Sure kill	•	term
SIGPIPE	13	Broken pipe	•	term

SAMP: Sun SPARC and SPARC64 (S), HP/Compaq/Digital Alpha (A), MIPS (M), and HP PA-RISC (P)

Important Signals

- SIGABRT
  - When a process calls *abort()*, this signal is delivered to the process. Terminates with a core dump.
- SIGALRM
  - Kernel generates this signal upon the expiration of a timer set by alarm() or settimer().
- SIGCHLD
  - Kernel generates this signal upon the termination of a child process.
- SIGHUP
  - When a terminal is disconnected, the controlling process of the terminal is sent this signal.
  - Daemons process repond to this signal by reinitializing from config file.

Important Signals

- SIGINT
  - Generated when a user presses Ctrl-c on a terminal to interrupt a process.
- SIGIO
  - Useful in signal driven I/O on sockets.
- SIGKILL
  - Can’t be blocked, ignored or caught by a handler. Always terminates a process. Used by admins.
- SIGPIPE
  - Kernel generates this when a pipe has no readers but a process writes into it.
- SIGQUIT
  - Generated when user presses Ctrl-\ on the terminal. It terminates the process with core dump.

Important Signals

- SIGSEGV
  - Generated when
    - Referencing an unmapped address
    - Updating a read-only page.
    - Accessing kernel memory.
- SIGSTOP
  - Used by admin to stop a process. Can't be ignored, blocked or handled. Process can be started again by SIGCONT signal.
- SIGTERM
  - Standard signal used for terminating a process. *init* process sends this signal during shutdown.
- SIGUSR1 & SIGUSR2
  - Kernel never generates these signals.
  - Available for programmer defined purposes.

Signals

- A signal is said to be *generated* by some event. Once generated signal is *delivered* to the process. Between the time it is generated and delivered, signal is said to be *pending*.
  - A pending signal is delivered as soon as the process scheduled to run or immediately if the process is running.
- Sometimes we do not want signals to be delivered to the process when executing a critical code segment.
  - Signals can be added to *signal mask* in the kernel. These signals are blocked. If a such is signal is generated, it will be kept pending.
  - When the mask is cleared, the pending signals are delivered to the process.

Signal Disposition

- A process can inform kernel how it wants to deal with a signal:
  - ignore/discard the signal (not possible with `SIGKILL` or `SIGSTOP`)
    - Kernel will not deliver such a signal to the process.
  - Catch the signal and execute a signal handler function, and then possibly resume execution.
    - Process installs a handler function for a signal with the Kernel. When the signal is received, kernel executes the function on the process behalf in user space.
  - Let the default action apply. Every signal has a default action.
    - Default action can be ignore, terminate the process. Depends on the signal.
- This choice is called the *signal disposition.*

Setting Signal Disposition

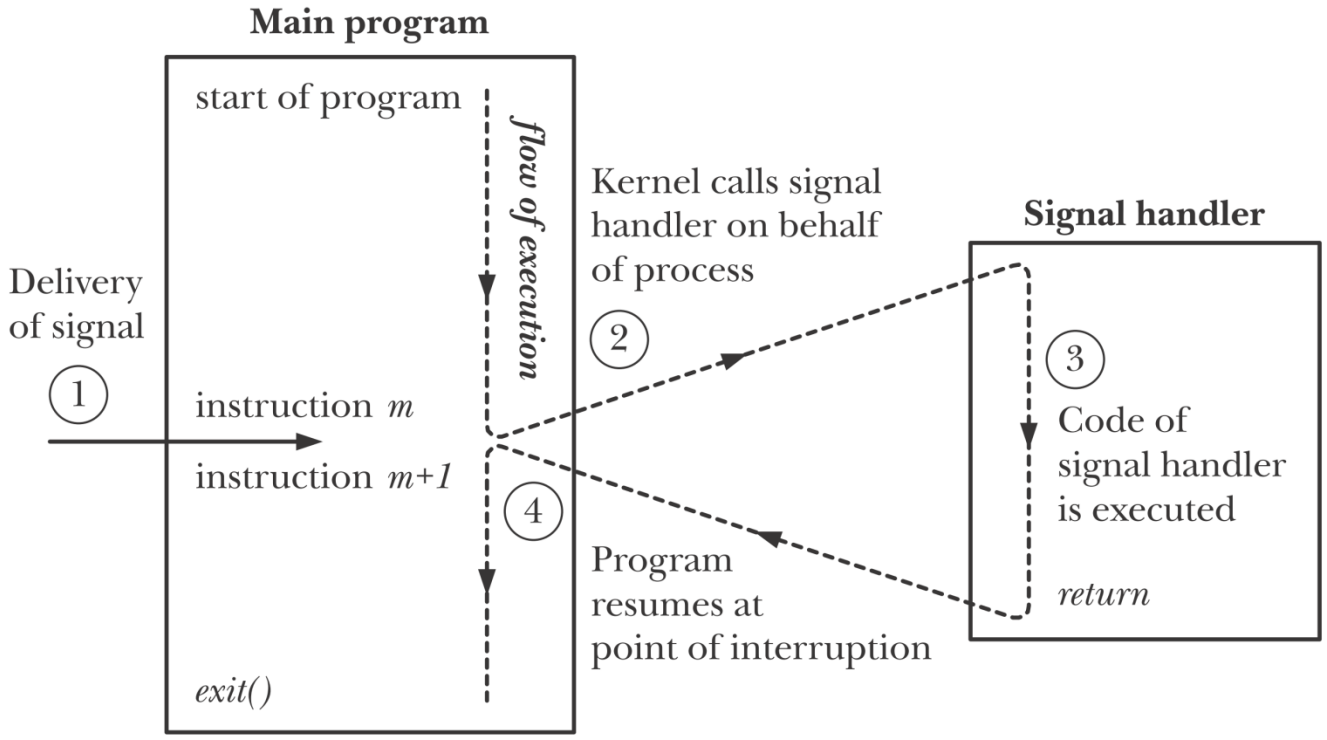
- *signal()* system call takes a function pointer *handler* and registers against the signal *signo*.

```
2   #include <signal.h>
3   typedef void Sigfunc(int);  /* my defn */
4   Sigfunc *signal( int signo, Sigfunc *handler );
5   /*Returns previous signal disposition if ok, SIG_ERR on error.*/
```

  - For other dispositions, the Sigfunc values are
    - SIG_IGN Ignore / discard the signal.
    - SIG_DFL Use default action to handle signal.
  - In case of error
    - SIG_ERR Returned by signal() as an error.
- We can't know the current disposition for a signal with this sys call.
  - *sigaction()* sys call can do that.

Signal Handlers

- Kernel calls the handlers on the process's behalf.
- Handler may be called at any time.
  - After the execution of the handler, program resumes from the point where it got interrupted.



Signal Handler: example

innovate achieve lead

```
 /*signals/ouch.c*/
#include <signal.h>
#include "tlpi_hdr.h"
static void
sigHandler(int sig)
{
    printf("Ouch!\n");                  /* UNSAFE (see Section R1: 21.1.2) */
}
int
main(int argc, char *argv[])
```

```
$ ./ouch
0                           Main program loops, displaying successive integers
Type Control-C
Ouch!                       Signal handler is executed, and returns
1                           Control has returned to main program
2
Type Control-C again
Ouch!
3
Type Control-\ (the terminal quit character)
Quit (core dumped)
```

kill() and raise()function

- Send a signal to a process (or group of processes).

```
2  #include <signal.h>
3  int kill( pid_t pid, int signo );
4  int raise(int signo);
5  /*Return 0 if successful, -1 on error.*/
```

  - pid > 0 send signal to process pid
  - pid== 0 send signal to all processes whose process group ID equals the sender's pgid.
    - e.g. parent kills all children
- Using raise(), a process can send a signal to itself.
- To know whether a PID is in use
  - Send a null signal to that PID
    - kill(PID, 0)
- kill command
  - kill -INT 9400

Signal Sets

- Many signal related system calls need a set of signals as input.
- Signal set is a data structure represented by *sigset_t* data type.
- Following are the library functions on *sigset_t* data type.

```
2  #include <signal.h>
3  int sigemptyset(sigset_t *set);
4  int sigfillset(sigset_t *set);
5  int sigaddset(sigset_t *set, int signo);
6  int sigdelset(sigset_t *set, int signo);
7  /*All four return: 0 on success, -1 on error */
8  int sigismember(const sigset_t *set, int signo);
9  /*Returns: 1 if true, 0 if false, -1 on error*/
```

Signal Mask

- For each process kernel maintains a *signal mask* – set of signals the process has blocked.
  - When a signal is to be delivered but it is blocked, then that signal is kept pending until it is unblocked by the process.
  - When a signal handler is invoked, the signal that caused invocation is automatically added to the mask.
- Using *sigprocmask()* system call, signals can be added or deleted from the mask or retrieve the mask.

```
2  #include <signal.h>
3  int sigprocmask(int  how , const sigset_t * set , sigset_t * oldset );
4  /*Returns 0 on success, or -1 on error*/
```

  - how is interpreted as
    - SIG_BLOCK: add *set* to the signal mask
    - SIG_UNBLOCK: delete *set* from the mask.
    - SIG_SETMASK: reset signal mask to *set.*

sigprocmask()

innovate achieve lead

```
 2  sigset_t blockSet, prevMask;
 3  /* Initialize a signal set to contain SIGINT */
 4  sigemptyset(&blockSet);
 5  sigaddset(&blockSet, SIGINT);
 6  /* Block SIGINT, save previous signal mask */
 7  if (sigprocmask(SIG_BLOCK, &blockSet, &prevMask) == -1)
 8      errExit("sigprocmask1");
 9  /* ... Code that should not be interrupted by SIGINT ... */
10  /* Restore previous signal mask, unblocking SIGINT */
11  if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
12      errExit("sigprocmask2");
```

Pending Signals

- To know pending signals we use *sigpending()* system call.

```
2  #include <signal.h>
3  int sigpending(sigset_t * set );
4  /*Returns 0 on success, or -1 on error*/
```

  - Pending signals are returned through *set.*
  - They are examined using *sigismember()* function.
- If a blocked signal is generated more than once then in most systems the signal is delivered only once. That is the signal is not queued.
- If many signals of *different* types are ready to be delivered (e.g. a `SIGINT`, `SIGSEGV`, `SIGUSR1`), they are not delivered in any fixed order.

Setting Signal Disposition: *sigaction()*

innovate achieve lead

```
2  #include <signal.h>
3  int sigaction(int  sig , const struct sigaction * act ,
4  struct sigaction * oldact );
5  /*Returns 0 on success, or -1 on error*/
6
7  struct sigaction {
8      void    (*sa_handler)(int);/* Address of handler */
9      sigset_t sa_mask;          /* Signals blocked during handler
10                                    invocation */
11     int      sa_flags;         /* Flags controlling handler invocation */
12     void    (*sa_restorer)(void);/* Not for application use */
13 };
```

- An alternative to *signal()*
  - Allows us to retrieve the disposition of a signal without changing it.
  - To atomically block signals while executing in a handler.
  - To retrieve attribute of a signal such pid, uid etc using SA_SIGINFO.
  - To automatically restart a system call using SA_RESTART.

Waiting for a Signal

- Calling *pause()* suspends execution of the process until the call is interrupted by a signal handler.

```
2  #include <unistd.h>
3  int pause(void);
4  /*Always returns -1 with errno set to  EINTR*/
```

  - When a signal is handled, *pause()* is interrupted and always returns -1.

- *sigsuspend():* atomically unblocks signals and suspends execution of the process until a signal is caught and its handler returns.

```
2  #include <signal.h>
3  int sigsuspend(const sigset_t * mask );
4  /*(Normally) returns -1 with errno set to  EINTR*/
```

  - Replaces the signal mask in the kernel with *mask.*
  - Suspends execution until signal handler returns.

pause() example

innovate achieve lead

```
 2  #include <signal.h>
 3  long n;
 4  void sigalrm(int signo){
 5          alarm(1);
 6          n=n+1;
 7          printf("%d seconds elapsed\n",n);
 8          }
 9  main(){
10          signal(SIGALRM, sigalrm);
11          alarm(1);
12
13          while(1) pause();
14          }
```

Synchronously Waiting for a Signal

- *pause()* and *sigsusepnd()* wait until a signal handler is executed. But we can do away with signal handlers.

```
2  #define _POSIX_C_SOURCE 199309
3  #include <signal.h>
4  int sigwaitinfo(const sigset_t * set , siginfo_t * info );
5  /*Returns number of delivered signal on success, or -1 on error*/
```

  - Suspends execution until one of the signals in *set* becomes pending and returns that signal number.
  - Useful only if signal is *set* are blocked using *sigprocmask().*
  - Faster than *sigsuspend()* because there is no signal handler.

sigwaitinfo() example

innovate achieve lead

```
 1  /*signals/t_sigwaitinfo.c*/
 2 /* Block all signals (except SIGKILL and SIGSTOP) */
 3     sigfillset(&allSigs);
 4     if (sigprocmask(SIG_SETMASK, &allSigs, NULL) == -1)
 5         errExit("sigprocmask");
 6     printf("%s: signals blocked\n", argv[0]);
 7
 8      for (;;) {/* Fetch signals until SIGINT (^C) or SIGTERM */
 9         sig = sigwaitinfo(&allSigs, &si);
10         if (sig == -1)
11             errExit("sigwaitinfo");
12         if (sig == SIGINT || sig == SIGTERM)
13             exit(EXIT_SUCCESS);
14      }
```

Non-local goto in Signal Handler

- POSIX does not specify whether *longjmp()* will restore the signal context. If you want to save and restore signal masks, use *siglongjmp()*.
  - In a signal handler the signal that caused signal handler invocation is added to the signal mask. It will not be removed, if *longjmp()* is called.
- POSIX does not specify whether *setjmp()* will save the signal context. If you want to save signal masks, use *sigsetjmp()*.

```
2  #include <setjmp.h>
3  int sigsetjmp(sigjmp_buf env, int savemask);
4  /*Returns: 0 if called directly, nonzero if returning from a call to siglongjmp*/
5  void siglongjmp(sigjmp_buf env, int val);
```

Q&A

Next Time

innovate achieve lead

- Please read through R1: chapters 8-10, 23,34, 43-44

innovate achieve lead

BITS Pilani

Pilani Campus

Thank You