Signals

- Signals are software interrupts
- A signal corresponds to an event
  - It is **raised** by one process (or hardware) to call another process's attention to an event
  - It can be **caught** (or **ignored**) by the subject process
- Signals have names start with "**SIG**"
  - For example **SIGSTOP** or **SIGCONT**
  - If you press CTRL-C at the terminal the signal **SIGINT** is sent

    to terminate the currently running process

Three possible actions when receive a signal :

Tell the Kernel:

- To ignore: Works for most signals.

  Does not work for **SIGKILL and SIGSTOP,** these signals terminate or stop the process and give the super user full control over stopping any running process

- To reinstate default : Default action applies.

  All signals have a default action..

Three possible actions when receive a signal : cont.

- To catch: Tell the kernel to invoke a given function whenever signal occurs

  Example: Write signal handler for **SIGTERM** to clean up after program is terminated

Catching a Signal: signal handler

defining signal handlers:

```
#include <signal.h>

        void (*signal (int signo, void (*func) (int) ) )  (int);
```

In other word: The function signal takes two arguments:

An integer and a pointer to a function that takes an integer and returns nothing

The function signal itself returns a pointer to a function that takes an integer as argument and returns nothing

func : is called signal handler

Catching a Signal : signal handler cont.

- The **signal()** function chooses one of three ways to handle the signal **signo**.

✓ If the value of **func** is SIG_IGN, the signal shall be ignored.

✓ If the value of **func** is SIG_DFL, default handling shall occur.

✓ Otherwise (catch), the application ensures that **func** points to a function to be called when that signal occurs. This function is called a "signal handler".

Example: signal(signo, SIG_DFL);

Code Example :

```
#include <signal.h>
#include <stdio.h>

void handler (int signal) {
  switch (signal) {
  case SIGUSR1: { printf ("SIGUSR1 received\n"); break; }
  case SIGUSR2: { printf ("SIGUSR2 received\n"); break; }
  case SIGINT: { printf ("SIGINT received\n"); break; }
  }
}

int main () {
  if (signal  (SIGUSR1, handler) == SIG_ERR)
    printf ("error\n");
  if (signal  (SIGUSR2, handler) == SIG_ERR)
    printf ("error\n");
  if (signal  (SIGINT, handler) == SIG_ERR)
    printf ("error\n");

  while (1)
    sleep (5);
}
```

- **SIGUSR1** and **SIGUSR2** not used by the system and can be used for any user-specific purpose
- We can use system command “**Kill**” to send this to our process

```
>kill –USR1 4720
```

- Normally, signals are set to their default action unless the process that call exec is ignoring the signal

  ex: background process (&) the shell will set interrupt

  and quit to be ignored

- Unix offers **sigaction** to set the signal handler

- Example signals and their integer values:

SIGALARM	=	14
SIGKILL	=	9
SIGTERM	=	15
SIGINT	=	2

To send signals system calls **kill** or **raise** can be used

```
int  kill (pid_t pid, int signo);
int  raise (int signo);
```

four conditions for the pid (process id) assignment to **kill**

pid > 0 : signal is sent to the process with that id.

pid == 0 : signal is sent to all processes share same group id with sender.

pid < 0 : signal is sent to all processes that have group id equals the absolute value of the pid.

pid ==-1 : undefined

-System call **raise** always send signal to calling process

-System call **alarm** can be used as a timer

```
unsigned int alarm(unsigned int seconds);
```

it returns 0 or the number of seconds until the previously set alarm

after the time is up the process get the **SIGALARM** sent. It may take

additional time due to scheduling delay

-System call **pause** can be used to stop the current process until a signal is caught

```
int pause(void);
```

it returns only if a signal handler is installed and the handler returns,

in that case it returns -1

Threads

- A thread is basic unit of cpu utilization
- It comprises :

  - thread ID

  - program counter

  - register set

  - a stack
- It shares with other threads belonging to the same process its

  - code section

  - data section

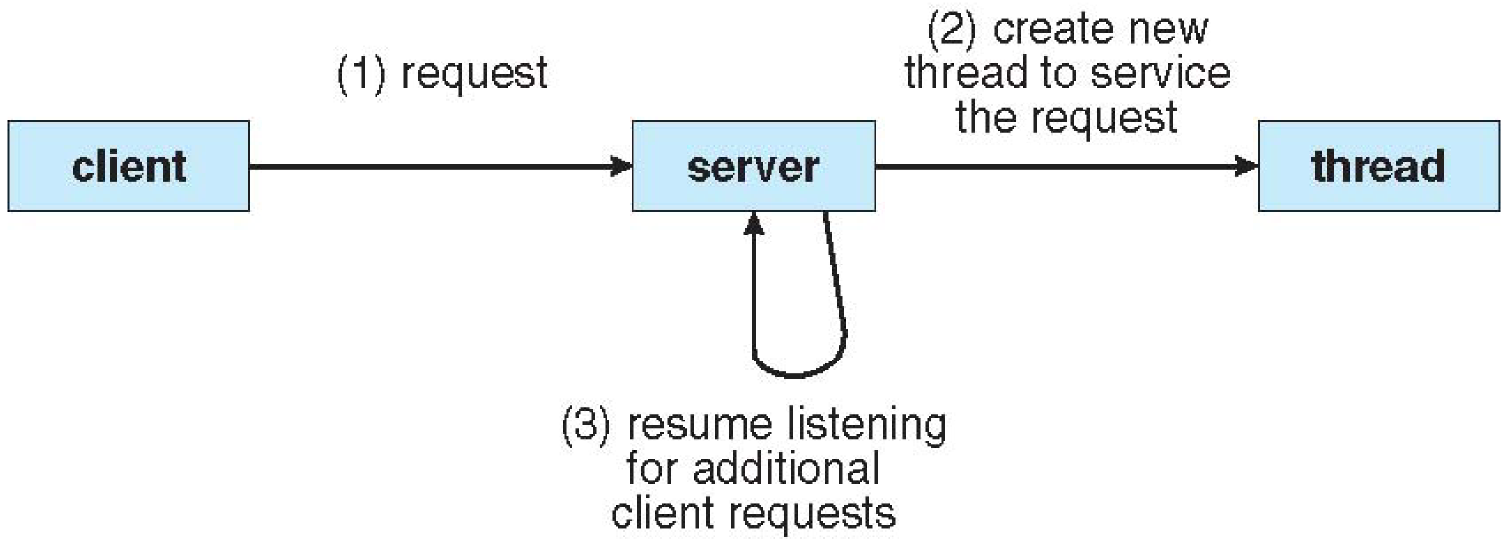
  - OS resources (open files and signals)

- Traditional process has a single thread of control,

  if it has multiple threads of control it can perform

  more than one task at a time

Multithreaded Server Architecture


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

Benefits of multithreading

- Responsiveness: Multithreading an interactive application may allow a program to continue running if part of it is blocked or performing a lengthy operation
- Resource Sharing : By default threads share memory and resources allows application to have several threads of activity within same address space

Benefits

- Economy: allocating memory and resources is costly
  since threads share resources, it is more economical to create context-switch threads
  it is more time consuming to mange process than threads

- Utilization of an Multiprocessor Architecture: to take advantage of the multiprocessor system where threads can be running in parallel

Creating Threads

- Depending on the OS API's exist for creating threads
  common ones are:
  - POSIX Pthreads on Unix
  - Win32 in Windows

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Linux)

phreads

```
Void  *runner ( void *param) ;  /* runner is function we create the thread for */
Pthread_t  tid;                 /* thread ID*/
Int param;                      /* parameters for runner*/
 pthread_create ( &tid ,NULL, runner,(void *)&param );
```

-Note that NULL means use default attributes

WIN32 threads

WIN32 thread creation is similar (text book p133,p135)

```
HANDLE threadhandle;
DWORD threadID;
DWORD WINAPI runner (void *param);
threadhandle = CreateThread (NULL,
                             0,
                             runner,
                             (void *)this,
                             0,
                             &threadID);
```

Threading Issues

- Does **fork()** duplicate only the calling thread or all threads?
- Using **fork()** copies the process as separate duplicate
- Some UNIX systems have 2 versions of **fork()**
  one duplicates all threads and another to duplicate only
  the thread invoked **fork()**
- If the thread invoked **exec()** the program in the parameter
  to **exec**() will replace the entire process including all threads

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Deferred cancelation may be preferable because it allows to free up resource otherwise system will not clam all resources from canceled thread

exp: open files will be closed only on the termination

of the process

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process