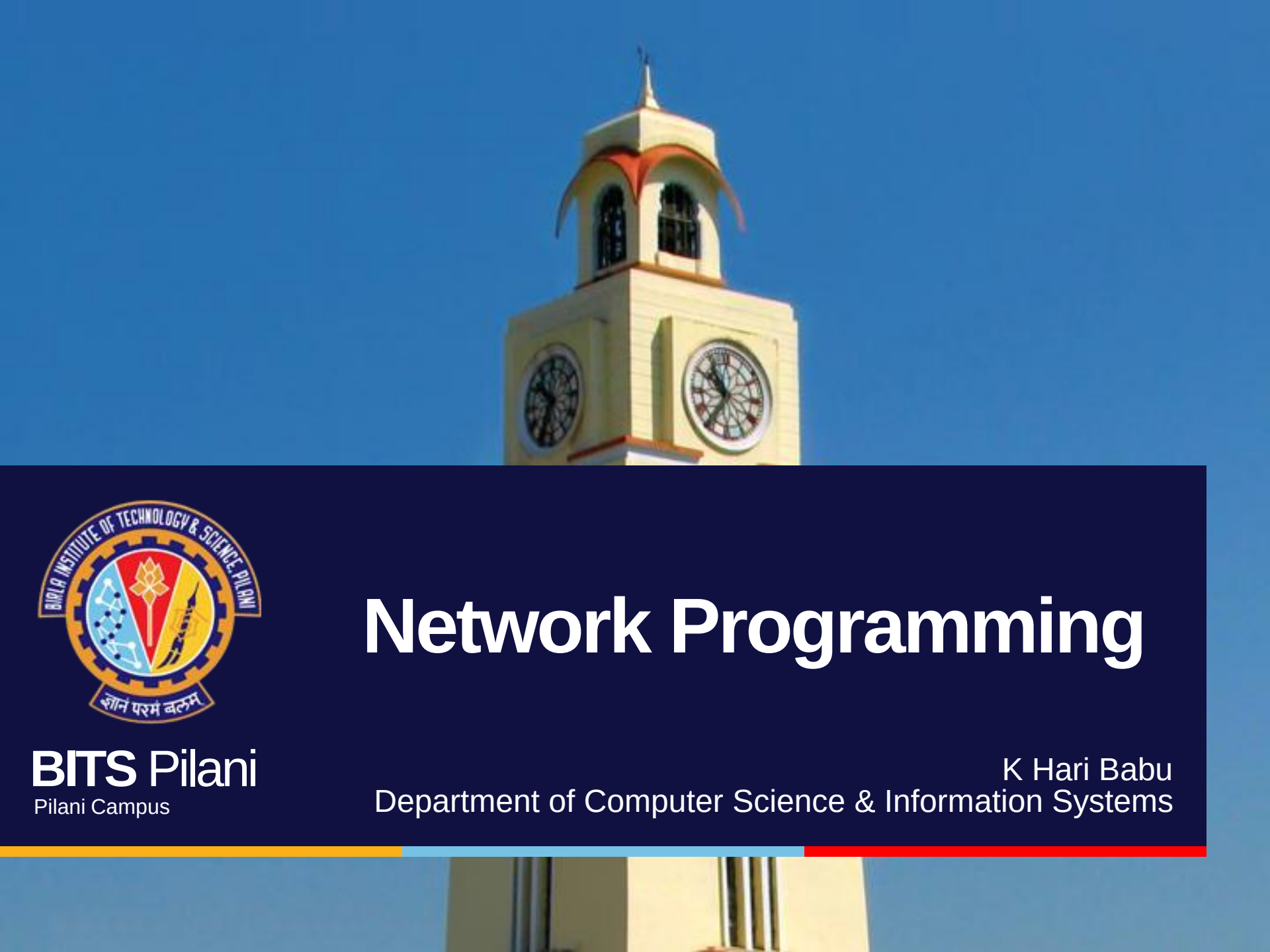


Network Programming

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

Outline

- Signals
- Time
  - Calendar time
  - Process time
- Timers
- File Permissions
- Process Credentials
- Process Groups

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

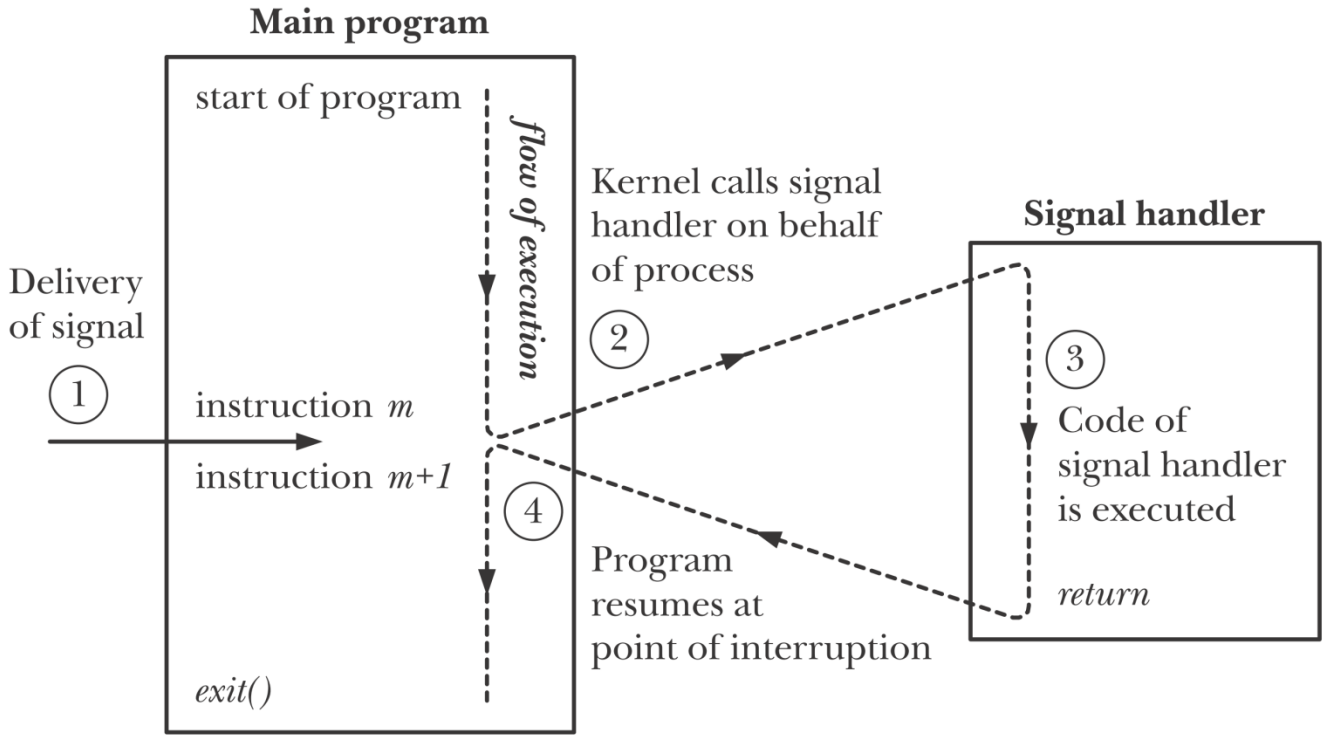
- *signal()* system call takes a function pointer *handler* and registers against the signal *signo*.

```
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 in *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);
```

innovate achieve lead

BITS Pilani

Pilani Campus

Time (R1: Ch10)

Time

- Within a program we may be interested in two kinds of time:
  - *Real time*: this is the time as measured from some standard point (calendar time) or from the start of a program (elapsed time).
    - Calendar time is useful for time stamping records.
    - Elapsed time is useful in taking periodic actions.
  - *Process time*: amount of CPU time used by a process.
    - Process time is useful for checking or optimizing the performance of a program or algorithm.
- Built-in hardware clock driven by battery enables the kernel to measure real and process time.

Hardware Clock & Software Clock

- *Hardware clock*: Driven by battery.
  - Stores Year, Month, Day, Hour, Minute, and the Seconds.
- *Software Clock*: The Linux kernel keeps track of time independently from the hardware clock.
  - Stores as the number of seconds since midnight January 1st 1970, UTC
  - During the boot, Linux sets its own clock to the same time as the hardware clock. After this, both clocks run independently.
    - Linux maintains its own clock because looking at the hardware is slow and complicated.
  - Resolution of the software clock: *jiffies.* Size of *jiffy* is defined by the constant HZ in kernel. It can be 100, 250, 1000 hertz, giving jiffy values of 10, 4, 1 millisecond.
    - The Linux kernel keeps track of the system clock by counting timer interrupts. Timer interrupts at every *jiffy* unit.

Calendar Time

- *gettimeofday()* system call returns the calendar time in the buffer pointed by *tv.*

```
2  #include <sys/time.h>
3  int gettimeofday(struct timeval * tv , struct timezone * tz );
4  /*Returns 0 on success, or -1 on error*/
5
6  struct timeval {
7      time_t      tv_sec;     /* Seconds since 00:00:00, 1 Jan 1970 UTC */
8      suseconds_t tv_usec;    /* Additional microseconds (long int) */
9  };
```

  - *tz* argument is obsolete now. This system call returns time in *timeval* struct.

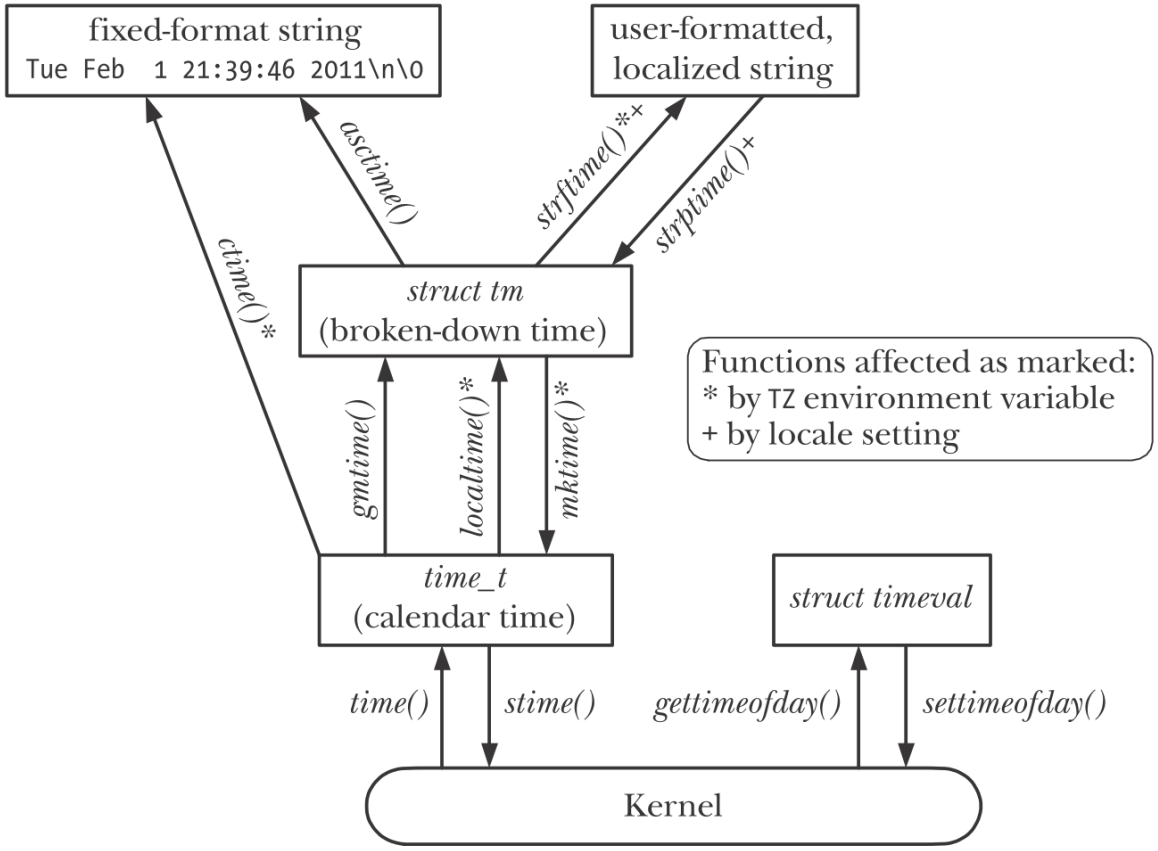
- *time()* system call: seconds elapsed since the Epoch.

```
2  #include <time.h>
3  time_t time(time_t * tp );
4  /*Returns number of seconds since the Epoch,or (time_t) -1 on error*/
```

  - General usage: *time(NULL)*

Library Functions

- Kernel gives only the time elapsed since the Epoch. Timezone info, date formats etc are specific to the process.



Process Time

- Process time is the amount of CPU time used by a process since it was created.
- For recording purposes, the kernel separates CPU time into the following two components
  - User CPU time: time spent in user mode
  - System CPU time: time spent in kernel mode.
- Time command gives CPU time for a command.

```
2  $ time ./myprog
3  real    0m4.84s
4  user    0m1.030s
5  sys     0m3.43s
```

times() system call

- times() system call returns process CPU time.

```
2  #include <sys/times.h>
3  clock_t times(struct tms * buf );
4  /*Returns number of clock ticks (sysconf(_SC_CLK_TCK)) since
5  "arbitrary" time in past on success, or (clock_t) -1 on error*/
6
7  struct tms {
8      clock_t tms_utime;   /* User CPU time used by caller */
9      clock_t tms_stime;   /* System CPU time used by caller */
10     clock_t tms_cutime;  /* User CPU time of all (waited for) children */
11     clock_t tms_cstime;  /* System CPU time of all (waited for) children */
12 };
```

  - *clock_t* values need to be divided by *sysconf(_SC_CLK_TCK) (usually 100)* to obtain seconds.

innovate achieve lead

BITS Pilani

Pilani Campus

Timers and Sleeping (R1: Ch23)

Timer

- A timer allows a process to schedule a notification for itself to occur at some time in the future.

```
2  #include <sys/time.h>
3  int setitimer(int  which , const struct itimerval * new_value ,
4                struct itimerval * old_value );
5  /*Returns 0 on success, or -1 on error*/
```

```
2  struct itimerval {
3      struct timeval it_interval;     /* Interval for periodic timer */
4      struct timeval it_value;        /* Current value (time until
5                                         next expiration) */
6  };
```

  - *which*: Using *setitimer()* a process can establish 3 types of timers.
    - *ITIMER_REAL:* counts down in real time. Generates SIGALRM.
    - *ITIMER_VIRTUAL:* counts down in virtual time (user mode CPU time). Generates SIGVTALRM signal.
    - *ITIMER_PROF:* counts in process time. SIGPROF is generated.

Timer


```
2  #include <unistd.h>
3  unsigned int alarm(unsigned int  seconds );
4  /*Always succeeds, returning number of seconds remaining on
5  any previously set timer, or 0 if no timer previously was set*/
```

- *alarm()* call is used to set a real-time timer in seconds. After expiry, it generates SIGALRM signal.
- alarm(0) cancels the existing timer.
- In Linux, there can be only one timer per process. Among setitimer() and alarm(), only one timer can be running at a give time.

Sleep

- Sometime we want to suspend the execution of a process for fixed amount of time.
  - This can be achieved by the combination of sigsuspend() and alarm().
- *sleep()* system call suspends execution for a specified seconds.

```
2  #include <unistd.h>
3  unsigned int sleep(unsigned int  seconds );
4  /*Returns 0 on normal completion, or number of
5  unslept seconds if prematurely terminated*/
```

  - *sleep()* can be interrupted by a signal. In that case it returns unfinished seconds.

High-resolution Sleeping

innovate achieve lead

```
2  #define _POSIX_C_SOURCE 199309
3  #include <time.h>
4  int nanosleep(const struct timespec * request , struct timespec * remain );
5  /*Returns 0 on successfully completed sleep,
6  or -1 on error or interrupted sleep*/
7
8  struct timespec {
9      time_t tv_sec;          /* Seconds */
10     long   tv_nsec;         /* Nanoseconds */
11 };
```

- Using *nanosleep()* sleep interval can be specified in nanoseconds.
- Buffer point by *remain* contains the unfinished sleep interval.
- Although interval can be specified in nanoseconds, it can't be finer than the software clock.

innovate achieve lead

BITS Pilani

Pilani Campus

Users, Groups, File Permissions and Process Credentials (R1: Ch8&9)

Users and Groups

- Every user has a unique login name and an associated numeric user identifier (UID).
- Each group also has a unique name and group identifier (GID).
  - A user can belong to one or more groups.
- /etc/passwd file
  - Login, encr passwd, userid, groupid, comment, home dir, login shell
- /etc/group file
  - Group name, encrypted password, group id, user list

```
2  /etc/passwd
3      avr:x:1001:100:Anthony Robins:/home/avr:/bin/bash
4  /etc/group
5      users:x:100:
6      staff:x:101:mtk,avr,martinl
7      teach:x:104:avr,rlb,alc
```

File Attributes

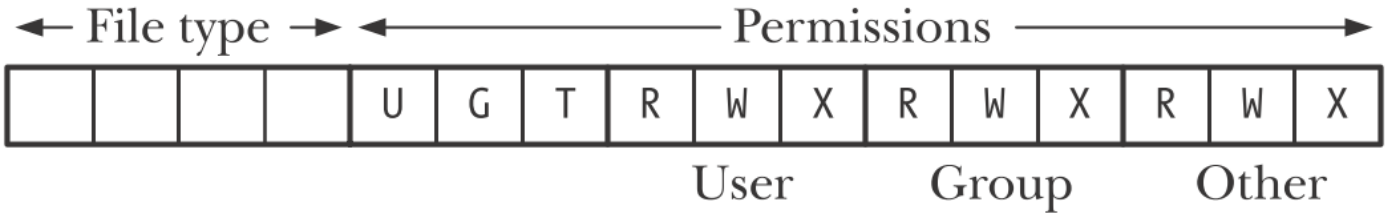
- File attributes can be obtained *stat()/fstat()* system calls.
  - They return the following structure.

```
 2  struct stat {
 3      dev_t     st_dev;       /* IDs of device on which file resides */
 4      ino_t     st_ino;       /* I-node number of file */
 5      mode_t    st_mode;      /* File type and permissions */
 6      nlink_t   st_nlink;     /* Number of (hard) links to file */
 7      uid_t     st_uid;       /* User ID of file owner */
 8      gid_t     st_gid;       /* Group ID of file owner */
 9      dev_t     st_rdev;      /* IDs for device special files */
10      off_t     st_size;      /* Total file size (bytes) */
11      blksize_t st_blksize;   /* Optimal block size for I/O (bytes) */
12      blkcnt_t  st_blocks;    /* Number of (512B) blocks allocated */
13      time_t    st_atime;     /* Time of last file access */
14      time_t    st_mtime;     /* Time of last file modification */
15      time_t    st_ctime;     /* Time of last status change */
16  };
```

    - *st_uid* and *st_gid* refers to ownership of the file.
    - *st_mode* refers to permissions.

File Types

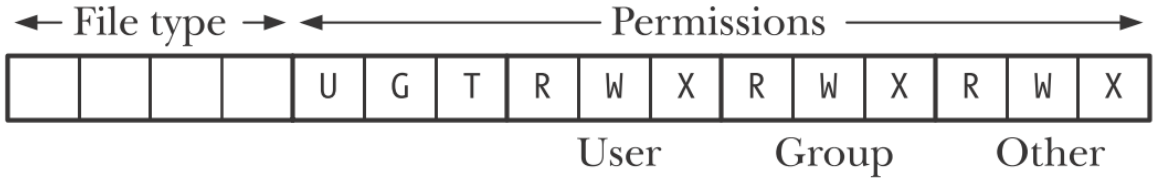
- *st_mode* is a bits mask serving the dual purpose of identifying the file type and specifying the file permissions.



  - File types

Constant	Test macro	File type
S_IFREG	S_ISREG()	Regular file
S_IFDIR	S_ISDIR()	Directory
S_IFCHR	S_ISCHR()	Character device
S_IFBLK	S_ISBLK()	Block device
S_IFIFO	S_ISFIFO()	FIFO or pipe
S_IFSOCK	S_ISSOCK()	Socket
S_IFLNK	S_ISLNK()	Symbolic link

File Permissions



- File permissions mask divides the permissions into 3 categories:
  - Owner (also known as user): The permissions granted to the owner of the file.
  - Group: The permissions granted to users who are members of the file's group.
  - Other: The permissions granted to everyone else.
- Three permissions may be granted to each category:
  - Read: the contents of the file may be read.
  - Write: contents may be changed.
  - Execute: file may be executed.
    - For scripts both read and execute permissions are required.

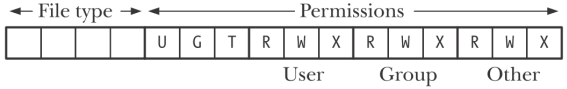
Permission on Directories

innovate achieve lead

- Directories have the same permission scheme as files. But they are interpreted differently.
  - Read: contents of the directory may be listed (e.g. *ls*)
  - Write: files may be created in and removed from the directory.
    - Not necessary to have any permission on the file itself in order to delete it.
  - Execute: files with in the directory may be accessed.
    - While accessing a file execute permission is required on all directories in the path.
    - Read permission only lets us see the list of files. Must have execute permission to access the contents or the info about the file.
    - If we have execute but not read permission, then we can access the file if we know the name of the file.
  - To add or remove files from the directory we need both execute and write permissions.

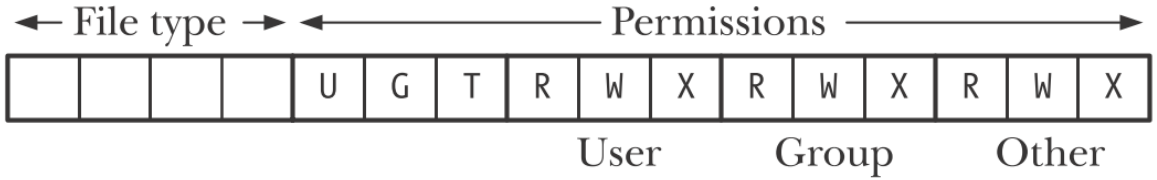
Permission Checking Algo

- The rules applied by the kernel when checking permissions are as follows:
  1. If the process is privileged, all access is granted.
  2. If the effective user ID of the process is the same as the user ID (owner) of the file, then access is granted according to the owner permissions on the file
  3. If the effective group ID of the process or any of the process supplementary group IDs matches the group ID (group owner) of the file, then access is granted according to the group permissions on the file.
  4. Otherwise, access is granted according to the other permissions on the file.



Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

Set-User-ID, Set-Group-ID and Sticky Bits



- A file apart from owner UID and GID, a file has set-user-id (U) and set-group-id (G) bits.
- When a set-user-id program is loaded by the kernel, it sets the effective user if of the process to be same as the user id of the executable file.
  - Similarly set group id
  - E.g. *passwd, mount, umount* etc
- Sticky bit (T): in older UNIX implementations, if a program file has sticky bit on it, it means copy of text segment was saved to swap area for faster execution.
  - Restricted deletion: On modern implementations, when a directory is set sticky, unprivileged process can delete a file only if it has write permission on the dir and also own the file or dir. E.g. /tmp directory.

Ownership and Permissions for a New File

- Ownership:
  - UID and GID of the new file are set same as the effective UID and effective GID of the process.
- Permissions:
  - Kernel uses the permissions specified in the *mode* argument to *open()* system call.
  - But these permissions are modified by the process file mode creation mask *umask*.
  - *umask*: aprocess attribute that specifies which permission bits should always be turned off when new files or dirs are created by the process.

umask() example

innovate achieve lead

```
1  #define MYFILE "myfile"
2  #define MYDIR  "mydir"
3  #define FILE_PERMS    (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)
4  #define DIR_PERMS     (S_IRWXU | S_IRWXG | S_IRWXO)
5  #define UMASK_SETTING (S_IWGRP | S_IXGRP | S_IWOTH | S_IXOTH)
6  int
7  main(int argc, char *argv[])
8  {
9      int fd;
10     struct stat sb;
11     mode_t u;
12     umask(UMASK_SETTING);
13     fd = open(MYFILE, O_RDWR | O_CREAT | O_EXCL, FILE_PERMS);
14     if (fd == -1)
15
16
17
18
```

```
1  $ ./t_umask
2  Requested file perms: rw-rw---- //This is what we asked for
3  Process umask:        ----wx-wx //This is what we are denied
4  Actual file perms:    rw-r----- //So this is what we end up with
5
6  Requested dir. perms: rwxrwxrwx
7  Process umask:        ----wx-wx
8  Actual dir. perms:    rwxr--r--
```

Process Credentials

- Every process has a set of associated UIDs and GIDs. These are referred to as process credentials.
- They are
  - Real user id and group id
  - Effective user id and group id
  - Saved set-user-id and saved set-group-id
  - Supplementary group ids.

Process Credentials

- Real user id and real group id
  - When a user login, the shell gets the real user id and real group id from the /etc/passwd file.
  - When a shell creates a process, real user id and effective user id are inherited form the parent (i.e. shell).
- Effective user id and effective group Id
  - They are used by the Kernel to check permissions for accessing a file.
  - Normally they are same as the real uid and gid respectively.
  - If the program file has set-user-id or set-group-id bits on, then effective uid and effective gid are set to owner uid and group gid of the program file.
- Saved set-user-id and saved set-group-id
  - These fields are used to copy the values from effective uid, gid.
  - Useful for writing secure programs: effective uid can be changed from real to saved user id or vice-versa during course of execution.

Retrieving Process Credentials

- The following system calls retrieve the real and effcetive uid and gids.

```
1  #include <unistd.h>
2  uid_t getuid(void);
3  /*Returns real user ID of calling process*/
4  uid_t geteuid(void);
5  /*Returns effective user ID of calling process*/
6  gid_t getgid(void);
7  /*Returns real group I
8  gid_t getegid(void);
9  /*Returns effective gr
```

  - They are always succe

- /proc/PID/status file c
  of the process.

```
1  Name:   vi
2  State:  R (running)
3  Tgid:   8415
4  Pid:    8415
5  PPid:   1965
6  TracerPid:      0
7  Uid:    1000    1000    1000    1000
8  Gid:    1000    1000    1000    1000
9  FDSize: 256
10 Groups: 1000 1001
11 .....
12 "/proc/self/status" [readonly] 38L, 764C
```

Modifying Effective IDs

innovate achieve lead

- *setuid()* system call changes the effective user ID to the value given by *uid*. Similarly *setgid()* changes the effective group id.

```
2  #include <unistd.h>
3  int setuid(uid_t  uid );
4  int setgid(gid_t  gid );
5  /*Both return 0 on success, or -1 on error*/
```

  - If an unprivileged process calls *setuid(),* then only the effective user id is changed.
    - *Uid* value must match with either real user id or saved set-user-ID.
  - If a privileged process (e.g. root) calls *setuid()* then all three i.e. real user id, effective user id and saved set-user-id are all set to *uid.*

innovate achieve lead

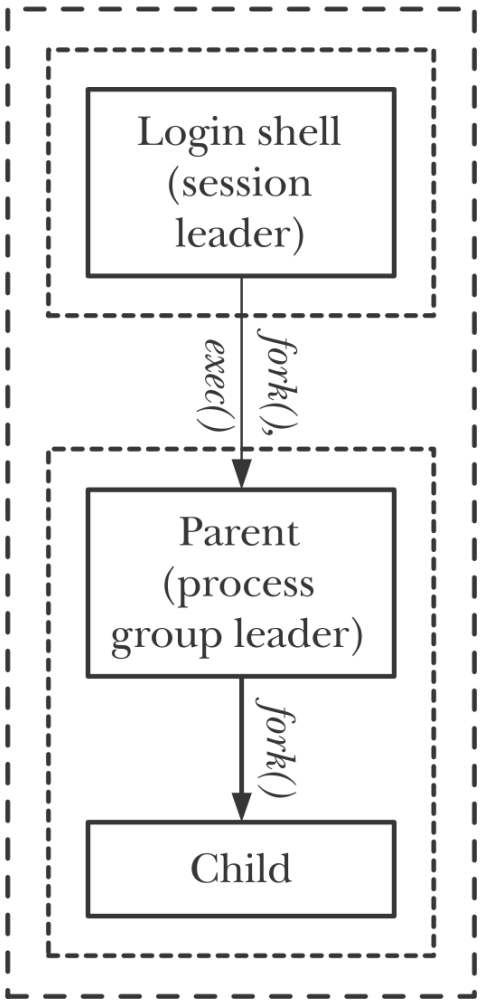
BITS Pilani

Pilani Campus

Process Groups, Sessions and Job Control (R1: Ch34)

Process Groups and Sessions

- Process groups and sessions form a two-level hierarchical relationship between processes.
  - Process group is a collection of related processes
  - Session is a collection of related process groups.
- Process groups and sessions are abstractions defined to support shell job control.
- *Process group* is synonymous with *job*.



Process Group

- A process group is a set of one or more processes sharing the same process group identifier (PGID).
- A process group has a *process group leader*, which is the process that creates the group and *whose process ID becomes the process group ID of the group*.
- A new process inherits its parent's process group ID.
- The process group leader need not be the last member of a process group.

```
2  $ echo $$                          //Display the PID of the shell
3  400
4  $ find / 2> /dev/null | wc -l &  //Creates 2 processes in background group
5  [1] 659
6  $ sort < longlist | uniq -c      //Creates 2 processes in foreground group
```

Session

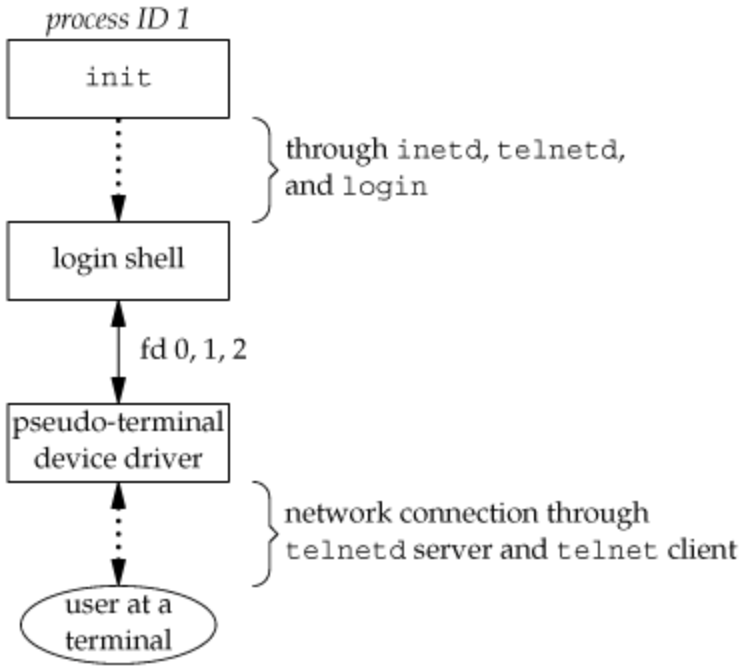
- A session is a collection of process groups.
  - A process's session membership is determined by its session identifier (SID).
- A session leader is the process that creates a new session and whose process ID becomes the session ID.
  - A new process inherits its parent's session ID.
- All of the processes in a session share a single controlling terminal.
  - The controlling terminal is established when the session leader first opens a terminal device.
  - A terminal may be the controlling terminal of at most one session.

Session & Process groups

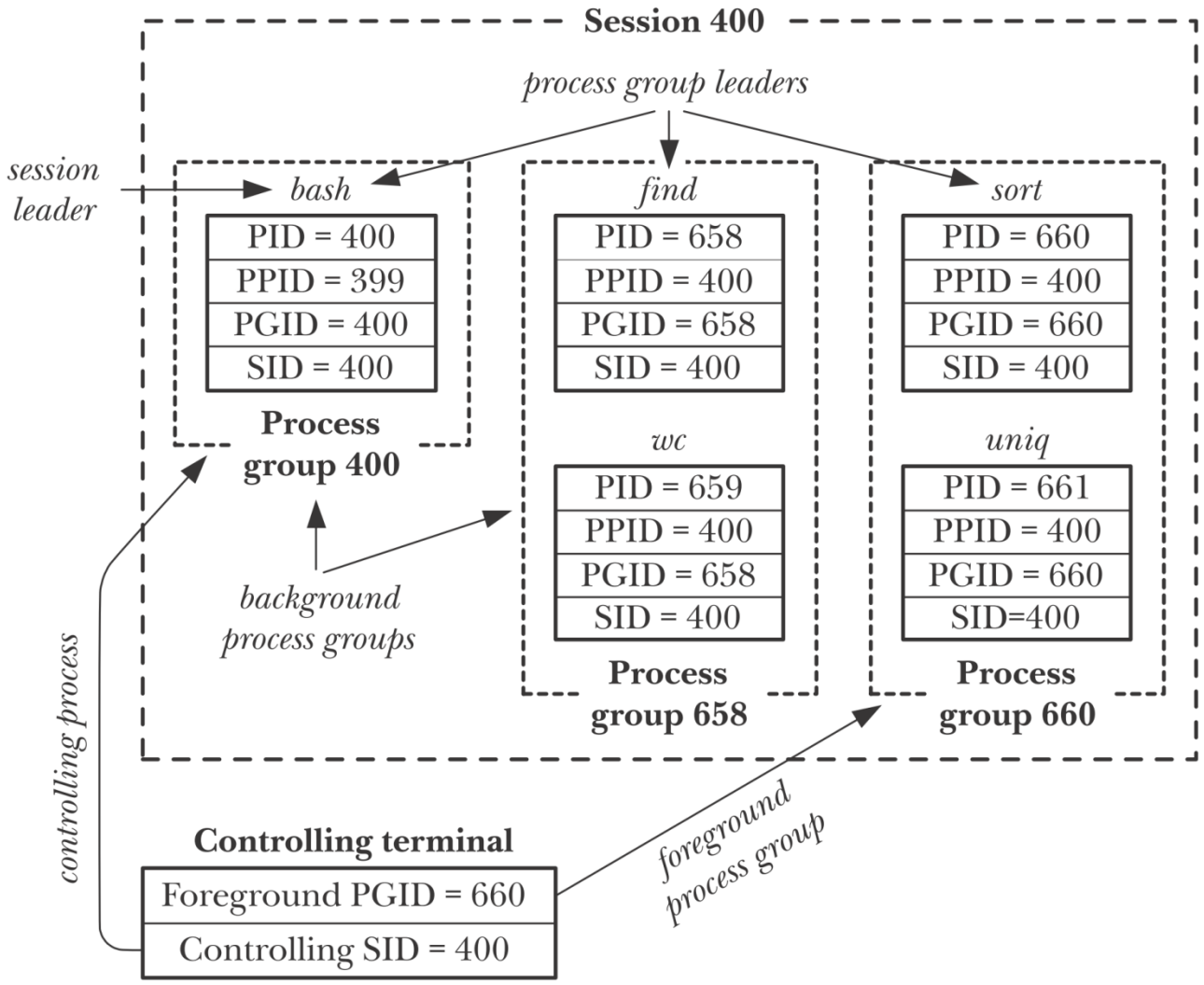
- One of the process groups in a session is the foreground process group for the terminal, and the others are background process groups.
  - Only processes in the foreground process group can read input from the controlling terminal.
  - When the user types one of the signal-generating terminal characters, a signal is sent to all members of the foreground process group.
    - Control-C which generates SIGINT
    - Control-\ which generates SIGQUIT
    - suspend character (usually Control-Z), which generates SIGTSTP .
- The session leader becomes the controlling process for the terminal.
  - kernel sends the session leader process a SIGHUP signal if a terminal disconnect occurs.

Network Login

- The *telnetd* process opens a pseudo-terminal device and sets up 0,1, 2 fds.
- execs *login* program.
- If we login correctly, it
  - changes to home dir
  - sets uid, gids and environment.
  - Execs login shell such as */bin/bash*.



```
$ echo $$                          //Display the PID of the shell
400
$ find / 2> /dev/null | wc -l &  //Creates 2 processes in background group
[1] 659
$ sort < longlist | uniq -c      //Creates 2 processes in foreground group
```



Process Groups

- Each process has a numeric process group ID that defines the process group to which it belongs.
  - A new process inherits its parent's process group ID. A process can obtain its process group ID using getpgrp().

```
1  #include <unistd.h>
2  pid_t getpgrp(void);
3  /*Always successfully returns process group ID of calling process*/
```

- The setpgid() system call changes the process group of the process whose process ID is pid to the value specified in pgid.

```
1  #include <unistd.h>
2  int setpgid(pid_t  pid , pid_t  pgid );
3  /*Returns 0 on success, or -1 on error*/
```

  - If *pid==pgid*, then a new process group is created else the process is moved to another process group.
  - Used by shell and login processes.

Session

- A session is a collection of process groups. The session membership of a process is defined by its numeric session ID.
  - A new process inherits its parent's session ID.
  - The getsid() system call returns the session ID of the process specified by pid.

```
2  #include <unistd.h>
3  pid_t getsid(pid_t  pid );
4  /*Returns session ID of specified process, or (pid_t) -1 on error*/
```

- If the calling process is not a process group leader, setsid() creates a new session.

```
2  #include <unistd.h>
3  pid_t setsid(void);
4  /*Returns session ID of new session, or (pid_t) -1 on error*/
```

setsid()

- When setsid() is called
  - The calling process becomes the leader of a new session, and is made the leader of a new process group within that session.
    - The calling process's process group ID and session ID are set to the same value as its process ID.
  - The calling process has no controlling terminal. Any previously existing connection to a controlling terminal is broken.
- The restriction why the caller can't be a process group leader
  - Because the process will be part of the new session with all its member processes being in another session.
  - This violates the two level hierarchy: session and process groups.

Job Control

- When we enter a command terminated by an ampersand(&), it is run as a background job.

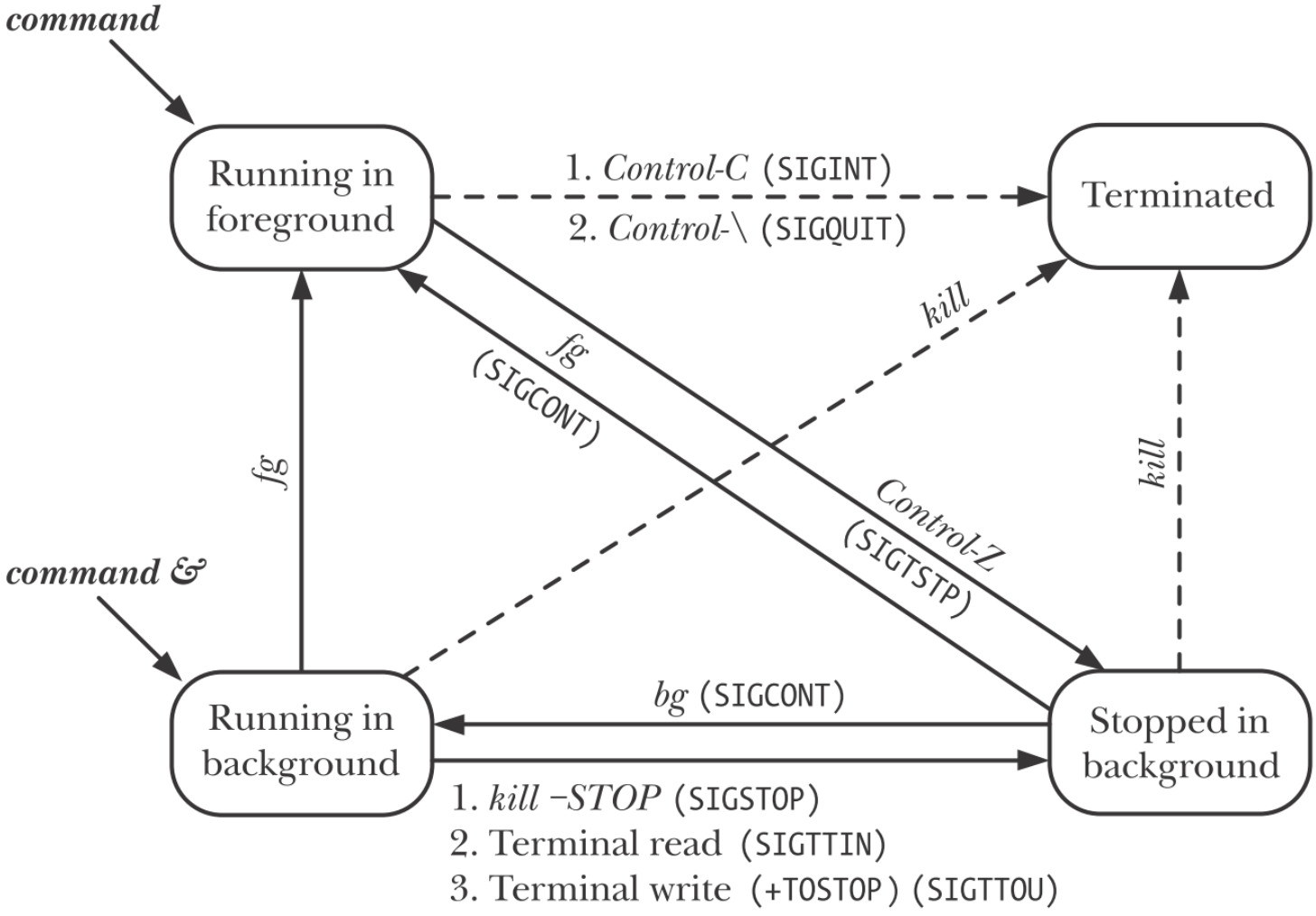
```
$ grep -r SIGHUP /usr/src/linux >x &
[1] 18932        //Job 1: process running grep has PID 18932
$ sleep 60 &
[2] 18934        //Job 2: process running sleep has PID 18934
```

  -

```
$ jobs
[1]- Running          grep -r SIGHUP /usr/src/linux >x &
[2]+ Running          sleep 60 &

$ fg %1
grep -r SIGHUP /usr/src/linux >x

Type Control-Z
[1]+ Stopped          grep -r SIGHUP /usr/src/linux >x

$ bg %1
[1]+ grep -r SIGHUP /usr/src/linux >x &
```

unique job

Job-control States



Q&A

Next Time

innovate achieve lead

- Please read through R1: chapters 43-44

innovate achieve lead

BITS Pilani

Pilani Campus

Thank You