Topics

1. GCC Toolchain
2. Shell Programming
3. File Operations
4. Advanced File Operations
5. Linux Kernel Compilation and build procedure
6. Process Management
7. Inter-Process Communication

GNU Toolchain

- GCC (GNU Compiler Collection)
- GNU Makefile
- GDB (GNU Debugger)
- Libraries:
- Procedure for creation of Static and Dynamic Libraries.
- Memory Layout of C program.

File Operations

- Linux File Structure
- Difference between System call and Standard Libraries
- /Proc and /Sys file Systems

Linux Kernel Compilation and build procedure

- Get Latest source code from Kernel
- Extrace source file
- Configure Kernel
- Compile Kernel
- Install Kernel
- Create on initrd image
- Update grub
- Reboot

Process Management

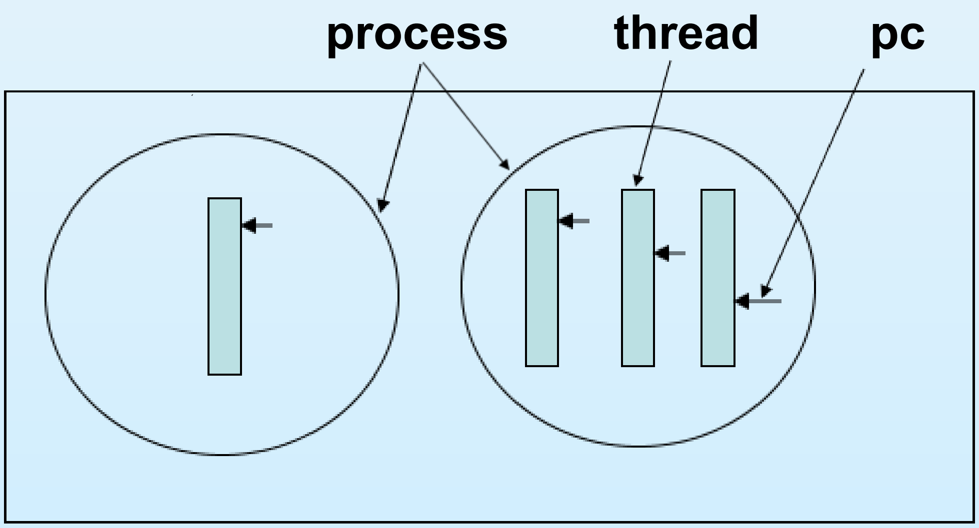
Processes

- Process Concept
  - Process ID & State
  - Process Tree
  - Process Control Block
  - Context Switching
  - Queues
  - Process Termination
  - Viewing Processes
- Process Scheduling
  - Linux Scheduling
  - RT Issues
- Process Creation
  - Replacing Process Image
  - Replication of Processes
  - Waiting for Processes

Process Concept

- Like files, a process is a fundamental abstraction in Unix/Linux
  - An executing instance of a program
- A process is an “an address space with one or more threads executing within that address space, and the required system resources for those threads.”
- The Linux kernel, supporting both preemptive multitasking and virtual memory, provides a process both a virtualized processor and a virtualized view of memory
- Each process consists of one or more *threads* of execution

- A thread is the unit of activity within a process, the abstraction responsible for executing code
- Each thread has
  - an id (*pid*)
  - a stack
  - state
  - program counter



Process Concept

- Parent processes create children processes, which, in turn create other processes, forming a tree of processes (*process tree*)
  - init process (pid=1) is the first created after booting.
- Resource sharing between parent and child processes
  - Parent and children share all resources
  - Children share subset of parent's resources (process share groups)
  - Parent and child share no resources
  - Resource sharing in Linux
    - File structure, resource limits, pipes, FIFOs etc
- Execution
  - Parent and children execute concurrently
  - Parent can wait until children terminate

Process ID (pid)

- Each process has a unique identifier, the *process ID* (maximum 32768)
- The process ID is represented by the pid_t type, defined in <*sys/types.h*>
- The getpid( ) system call returns the process ID of the invoking process
- The getppid( ) system call returns the ID of the parent of the invoking process

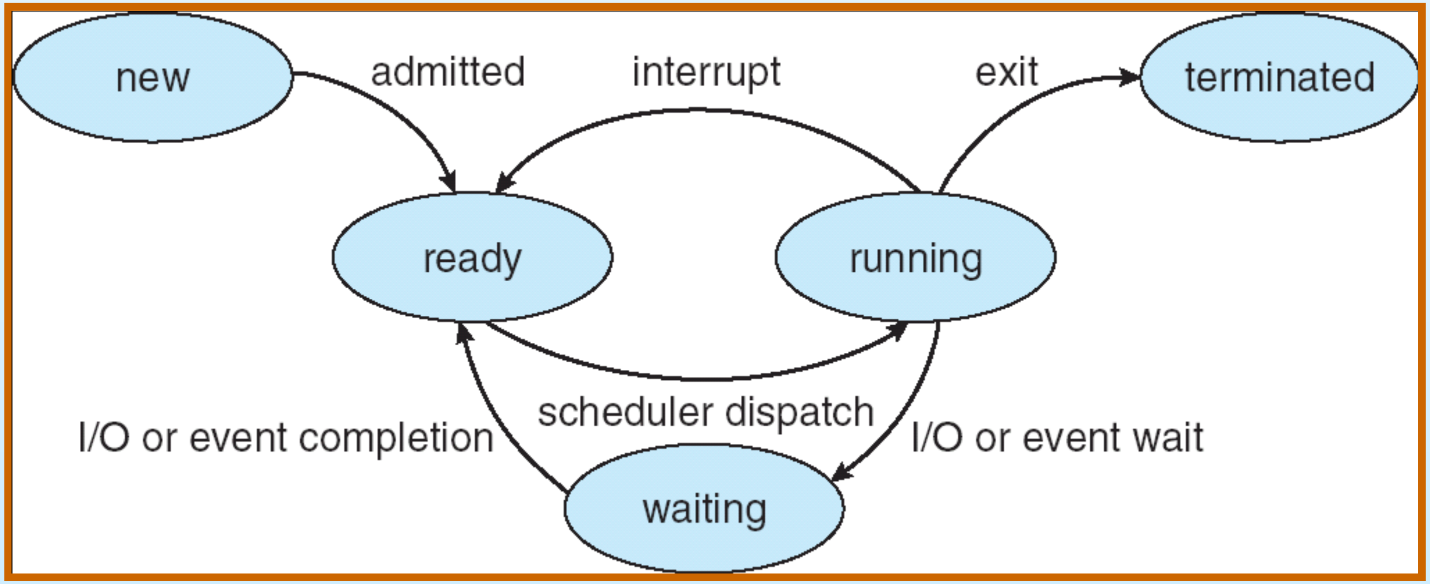
```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    printf ("My pid=%d\n", getpid ( ));
    printf ("Parent's pid=%d\n", getppid ( ));
    return (0);
}
```

My pid=6811

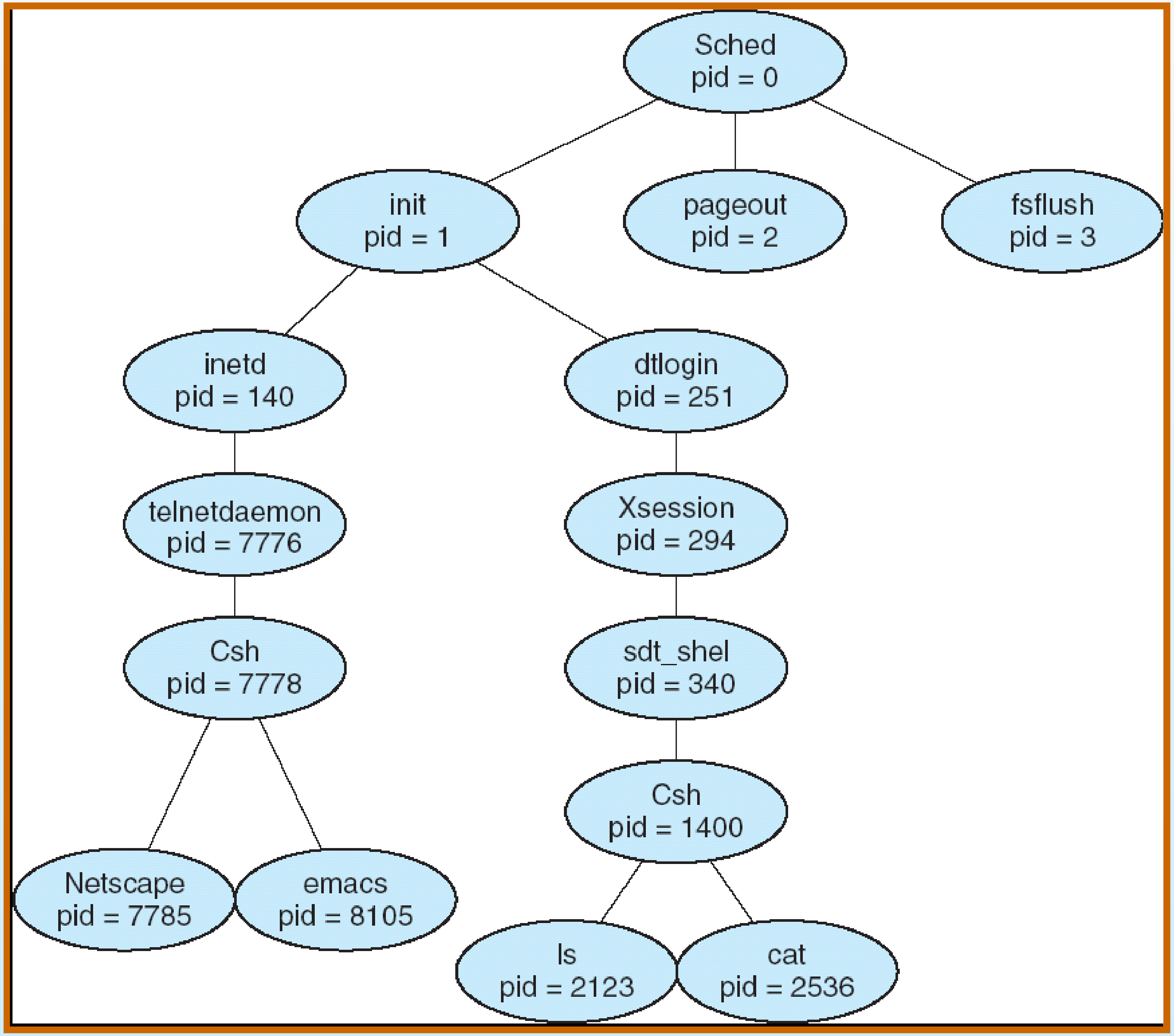
Parents pid=6723

Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution
- State values: TASK_RUNNING, TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_STOPPED, TASK_ZOMBIE



A tree of processes



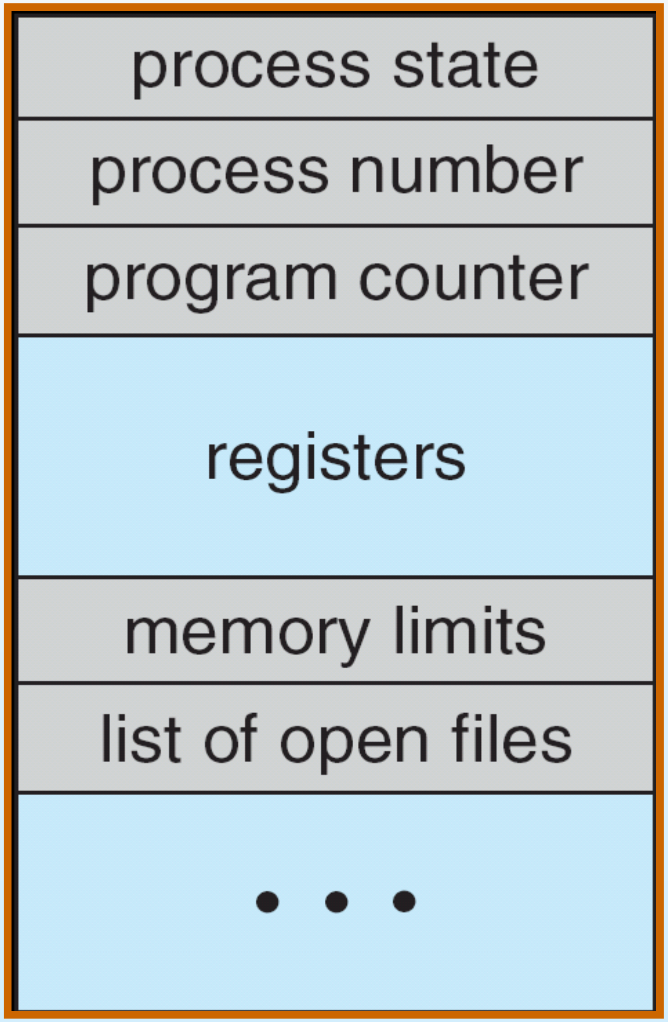
Init Process

- The first process that the kernel executes after booting the system, called the *init process*, has the pid 1
- The init process handles
  - The remainder of the boot process
  - Initializing the system
  - Starting various services
  - Launching a login program
- The Linux kernel tries four executables, in the following order:
  - */sbin/init*: The preferred and most likely location for the init process.
  - */etc/init*: Another likely location for the init process.
  - */bin/init*: A possible location for the init process.
  - */bin/sh*: The Bourne shell, if it fails to find an init process

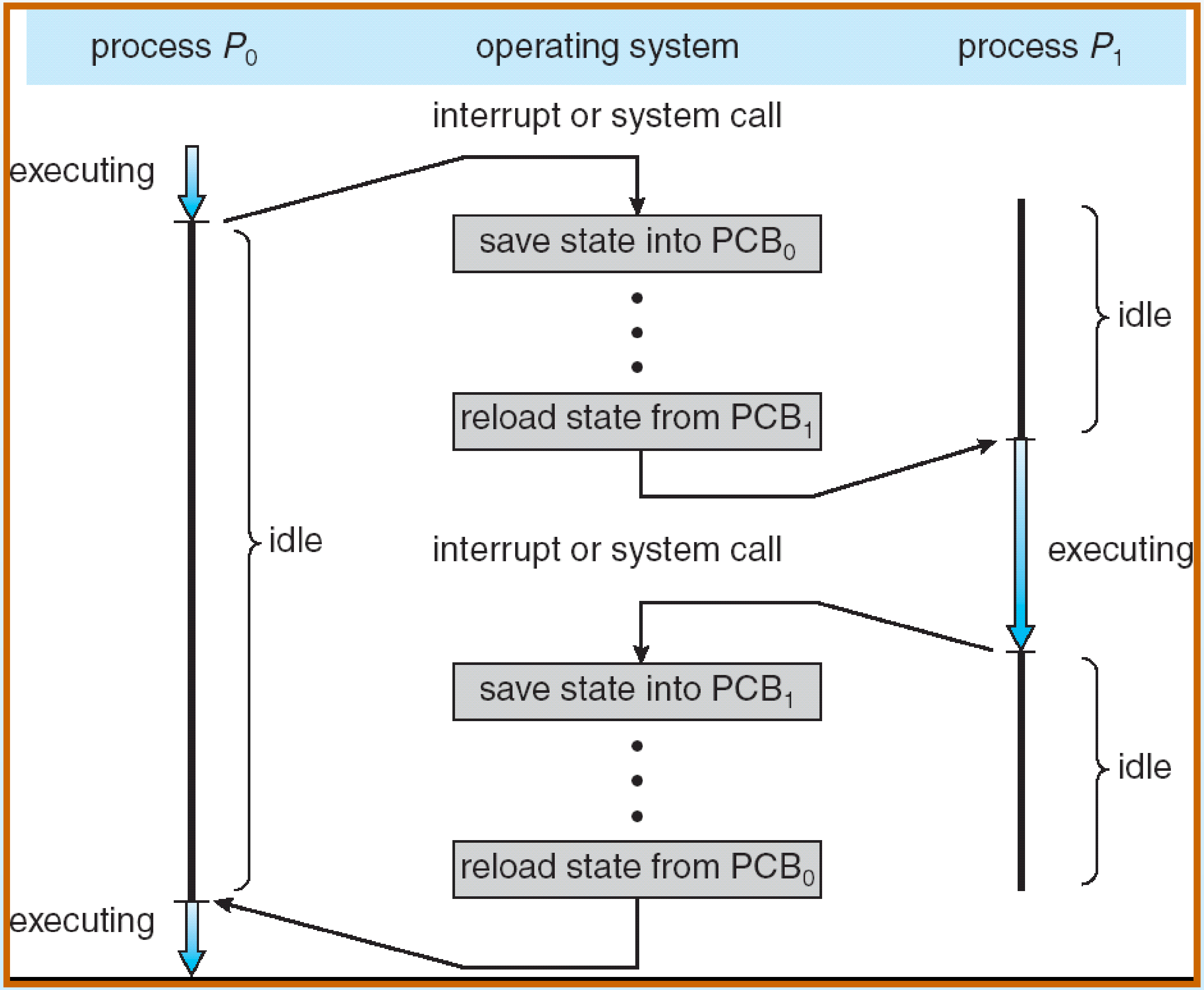
Process Control Block (PCB)

Information associated with each process stored in a block of memory known as PCB or Process Descriptor

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



CPU Switch From Process to Process



Process Termination

- Process executes last statement (**exit 0** for successful exit, **exit 1**, or >0 for exit with error condition) to inform the operating system to delete it
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
- Parents may wait (via **wait**) for a child process to terminate
  - If a child process terminates before the parent does wait, Linux does not delete it fully but keeps the exit information for the parent (**zombie)**
- If a parent process exits
  - Some operating system do not allow child to continue if its parent terminates
    - All children terminated - *cascading termination*
  - In Linux, if a parent terminates before a child, the child is re-parented to another process in the group or to the init process
- The library call exit() is a wrapper over the kernel syscall _exit(). exit() flushes pending I/O, closes file descriptors and does other cleanup (memory, semaphores, etc) before calling _exit()

Viewing Processes

- Linux Process Table
  - a data structure describing all of the processes that are currently loaded
- Viewing processes
  - The **ps** command shows the processes in the system or belonging to a user

  $ **ps -af**

  UID PID PPID C STIME TTY TIME CMD

  root 433 425 0 18:12 tty1 00:00:00 [bash]
- Process priority

  $ **ps -l**

  F S UID PID PPID C PRI NI SZ WCHAN TTY TIME CMD

  000 S 500 1362 1262 2 80 0 789 schedu pts/1 00:00:00 oclock

Scheduling Methods

- First-Come, First-Served (FCFS)
  - Non-preemptive; convoy effect
- Shortest-Job-First (SJF) Scheduling
  - Non-preemptive & Preemptive (SRTF) – optimal
  - Issue – weighted average prediction of length of next CPU burst
- Priority Scheduling
  - Starvation – increase priority of the process with time (Aging)
  - For equi-priority processes
    - FIFO
    - RR (Round Robin) – time slice (10-100 ms); context switch – 100μs
- Multilevel queues – (foreground/background) with own scheduling policy
- Multiple-Processor Scheduling
  - Asymmetric Multiprocessing – one CPU (master) handles scheduling
  - Symmetric Multiprocessing – each processor is self-scheduling
    - Processor affinity to improve cache performance
    - Load Balancing by Push and Pull migration

Linux Scheduling

- Preemptive, priority-based scheduling
- Two algorithms: time-sharing and real-time
- Time-sharing (nice values range from 100 to 140)
  - Dynamic priorities
    - I/O bound jobs have higher priorities
    - Aging adds priority
- Real-time (Real-time priorities vary from 0 to 99)
  - Soft real-time
  - Posix.1b compliant – (static priorities)
    - FCFS and RR

RT issues in Linux

- A real time system can be defined as a *"system capable of guaranteeing timing requirements of the processes under its control"*
  - Fast – low latency (quick response to external, asynchronous events)
  - Predictable – deterministic completion time of tasks with certainty
- Linux is a standard time-sharing operating system with good average performance and highly sophisticated services but lacks real time support
  - Changes in the interrupt handling
  - Scheduling policies
- Two Approaches to make Real Time OS
  - Develop new Linux like RTOS – QNX Neutrino, Vx Works, Lynx, µSoft CE
  - Plug-in a layer (Real-time kernel) underneath the Linux Kernel to handle interrupts and do real-time scheduling
- A real time kernel intercepts the interrupts, not allowing Linux to disable them, and decides what to despatch – schedule tasks or send to Linux
  - **RTLinux** was developed at the New Mexico Institute of Technology by Michael Barabanov and Professor Victor Yodaiken
  - **Real-Time Application Interface (RTAI)** was developed at the Dipartimento di Ingeneria Aerospaziale, Politecnico di Milano (DIAPM) by Professor Paolo Mantegazza

Starting new Processes

- **Starting New Processes**

```
// system1.c to create a new process by
//using the system library function
#include <stdlib.h>
#include <stdio.h>
int main()
{
    printf("Running ps with system\n");
    system("ps -ax");
    printf("Done.\n");
    exit(0);
}
// to run in background system2.c
// replace system("ps -ax") with
    system("ps -ax" &);
```

```
$ ./system1
Running ps with system
PID TTY STAT TIME COMMAND
- - -
1480 pts/1 S 0:00 ./system1
1481 pts/1 R 0:00 ps -ax
Done.
$ ./system2
Running ps with system
PID TTY STAT TIME COMMAND
1 ? S 0:05 init
2 ? SW 0:00 [keventd]
...
Done.
$ 1483 ? S 0:00 kdeinit: klipper
1484 pts/1 R 0:00 ps -ax
```

Replacing a Process image

- **exec** function replaces the current process with a new process specified by the path or file argument

  int execl (const char *path, const char *arg0, ..., (char *)0);

  int execlp (const char *file, const char *arg0, ..., (char *)0);

  int execle (const char *path, const char *arg0, ..., (char *)0, char *const envp[]);

  int execv (const char *path, char *const argv[]); //**basic syscall**

  int execvp (const char *file, char *const argv[]);

  int execve (const char *path, char *const argv[], char *const envp[]);

"l" indicates that the arguments are provided in a null terminated list; "v" in an array (vector);

"p" indicates the full PATH must be searched for the file;

- "e" indicates a new environment is also supplied for the new process
- **ret = execl("/bin/ps", "ps", "-ax", 0);** /* assumes ps is in /bin */

  \- replaces the current process image by loading the program pointed at by path
- **ret = execlp("ps", "ps", "-ax", 0);** /* assumes /bin is in PATH */

  \- To use the "v" or array option

  const char *args[ ] = { "ps", "-ax", NULL };
- **ret = execv ("/bin/ps", args);** or **ret = execvp ("ps", args);**

exec call

- A successful invocation of exec call does not return; it ends by jumping to the entry point of the new program, and the just-executed code no longer exists in the process' address space
- On error execl( ) returns -1, and sets errno to indicate the problem

  (examples of errno values: EACCESS, ENOEXEC, ENOMEM, etc)

  Note: errno variable is defined in <errno.h> include file
- On successful exec call
  - some properties of process are same: pid, priority, owning user and group
  - some properties change: signals, memory locks, statistics
  - open files are retained; generally these are closed before the exec call

Replacing a Process image

- pexeclp

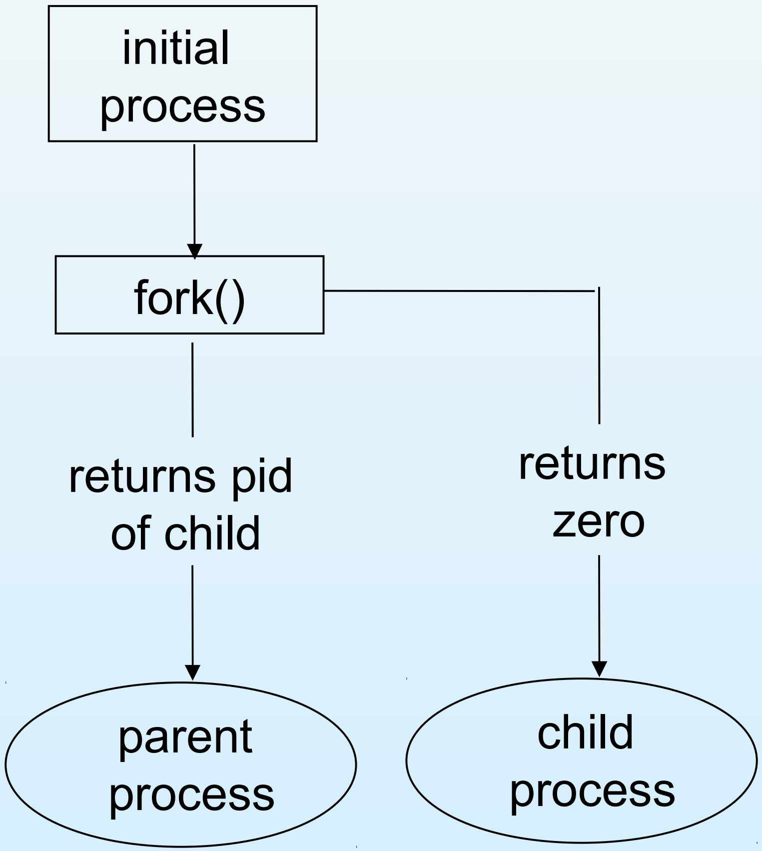
```
#include <unistd.h>
#include <stdio.h>
int main()
{
    int rtn;
      printf("Running ps with execlp\n");
      rtn = execlp("ps", "ps", "-ax", 0);
      if (ret == -1)
         perror ("execl");
      printf("Done.\n");
      exit(0);
}
```

```
$ ./pexeclp
Running ps with execlp
PID TTY STAT TIME
    COMMAND
1 ? S 0:05 init
2 ? SW 0:00 [keventd]
...
1262 pts/1 S 0:00 /bin/bash
1273 pts/2 S 0:00 su -
1274 pts/2 S 0:00 -bash
```

Duplicating a Process Image

- We can create a new process by calling **fork**. This system call duplicates the current process (creates a new entry in the process table with same attributes as the current process)
- Both processes continue from next instruction

```
pid_t new_pid;
- - -
new_pid = fork();
switch(new_pid) {
     case -1 : /* Error */
          break;
     case 0 : /* We are child */
        - - -
          break;
     default : /* We are parent */
        - - -
          break;
}
```



fork

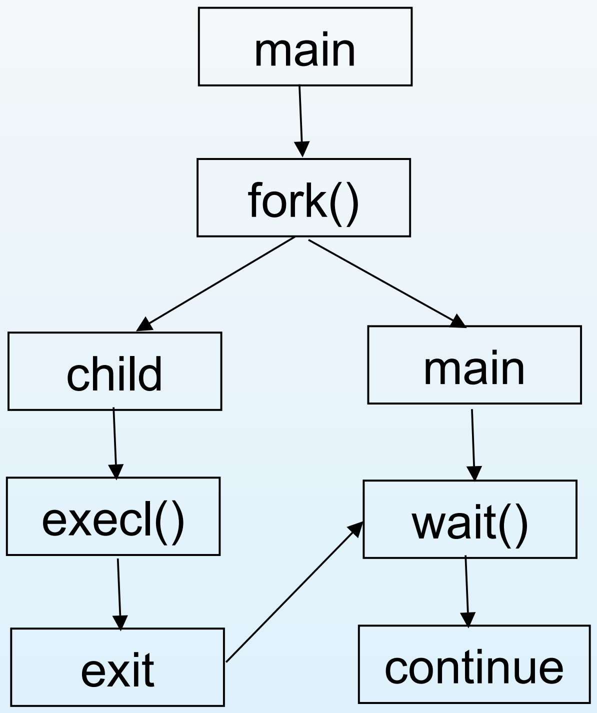
```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
 int main()
 {
      pid_t new_pid; char * msg; int n;
      printf (“fork program starting\n”);
      new_pid = fork();
      switch(new_pid) {
      case -1: //fail
           perror (“fork failed”); exit(1);
      case 0: //child
           msg = “child”; n=5; break;
      default: //parent
           msg = “parent”; n=3; break;
       }
      for (; n>0; n--) {
           puts(msg); sleep(1)
      }
      exit (0);
}
```

- $ ./fork1

```
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
$ This is the child
This is the child
```

fork and exec

```
#include <sys/wait.h>
int main()
{
pid_t  pid;
     /* fork another process */
     pid = fork();
     if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       exit(-1);
     }
     else if (pid == 0) { /* child process */
       execl ("/bin/ls", "ls", NULL);
     }
     else { /* parent process */
     /* parent will wait for the child to complete */
       wait (NULL);
       printf ("Child Complete");
       exit(0);
     }
}
```



fork call

- **The successful fork() call**
  - The fork() call makes a copy of the parent process structure for the child
    - Address space, resource limits, umask, controlling terminal, directory structure, current working directory, file pointers etc
  - The following will be different
    - PID, PPID, resource utilizations (child set to 0), signals etc
- **On failure**
  - fork() returns – 1, and error set in errno (EAGAIN, ENOMEM)

Waiting for a Process – wait()

- Parent process can wait for the child to finish by calling

  pid_t wait (int *stat_val);
- The call returns PID & exit status of the child process in stat_val
- Need macros to interpret

  WIFEXITED(stat_val) – Nonzero if the child is terminated normally

  WEXITSTATUS(stat_val) – child exit code If WIFEXITED is nonzero

  WIFSIGNALED(stat_val) – Nonzero if child terminated on uncaught signal

  WTERMSIG(stat_val) – signal number if WIFSIGNALED is nonzero
- To wait for a specific process

  pid_t waitpid (pid_t pid, int *status, int options);
  - Options WNOHANG – Do not block

Waiting for a Process – wait()

```
#include <sys/wait.h>
int main()   {
   pid_t pid;
  - - -
   pid = fork();
   switch(pid)
   {
   case -1:
     - - -
   case 0:
      printf("This is the child\n");
      wait(10);
   exit(0);
   default:
     - - -
   }
```

```
if (pid != 0) {     //Parent
    int stat_val;
    pid_t child_pid;
    child_pid = wait(&stat_val);
    printf("Child finished: PID = %d\n", child_pid);
    if(WIFEXITED(stat_val))
          printf("Child exited with code %d\n",
    WEXITSTATUS(stat_val));
    else
          printf("Child terminated abnormally\n");
    }
 exit(exit_code);
 }
```

```
$ ./wait
This is the child
Child finished: PID = 1582
Child exited with code 37
$
```

Process Priority

- Unix has historically called process priorities *nice values*
  - Legal nice values range from –20 to 19 inclusive, (default value of 0) (the lower a process' nice value, the higher its priority)
  - Linux provides system calls for retrieving and setting a process' nice value
- **int nice (int inc);**
  - If inc = 0, nice returns current value
  - For inc > 0, nice increments the nice value by inc & returns the new value
- getpriority(), setpriority(), renice()
  - Get and set priority for individual process, group or user

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    int ret, val;
    val = nice (0);
    printf (“current nice
value is %d\n”, val);
ret = nice (10); /* increase nice by 10 */
    if (ret == -1)
        perror ("nice");
    else
        printf ("nice value is now %d\n",
ret);
    return 0;
}
```

current nice value is 0
Nice value is now 10

Signals

- *Signals* are software interrupts for handling asynchronous events
  - External – eg. the interrupt character (Ctrl-C)
  - Internal – as when the process divides by zero
  - A process can also send a signal (“raise”) to another process
- Signal life cycle
  - A signal is “*raised*”
  - Kernel *stores* and *delivers the signal*
  - The process *handles* the signal
- Signal handling
  - Ignore the signal – 2 cannot be ignored – SIGKILL & SIGSTOP
  - Catch and handle the signal by registered functions (signal handlers)
    - SIGINT and SIGTERM are two commonly caught signals
  - Default action – terminate the process (result in core dump)

Signals

- Signals have identifiers

  defined in <signal.h>

  (They also have integers)
- Processes can handle a signal they catch by defining a signal function to handle them

  (void) signal(SIGINT, ouch);
- A successful call to signal( ) removes the current action taken on receipt of the signal SIGINT, and instead handles the signal with the signal handler specified by the handler function ouch(int)
- To reset the signal to the default behavior or to ignore by

  SIG_DFL – default handler

  SIG_IGN – ignore the signal

Signal Name	Description
SIGABORT	Process abort
SIGALRM	Alarm clock
SIGFPE	Floating-pt exception
SIGHUP	Hangup
SIGILL	Illegal Instruction
SIGINT	Terminal interrupt
SIGKILL	Kill
SIGPIPE	Write on pipe, with no reader
SIGQUIT	Terminal quit
SIGSEGV	Invalid memory segment access
SIGTERM	Termination (ctrl-c)

Signals

ctrlc1.c

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ouch(int sig)
{
    printf(“OUCH! - I got signal %d\n”, sig);
    (void) signal(SIGINT, SIG_DFL);
}
int main()
{
    (void) signal(SIGINT, ouch);
    while(1) {
     printf(“Hello World!\n”);
     sleep(1);
    }
}
```

```
$ ./ctrlc1
Hello World!
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
^C
$
```

Signals

- A child process does not inherit the signal handlers from the parent
- They are set to default handlers
- A process may send a signal to another process with same user ID, including itself, by calling kill.

  #include <sys/types.h>

  #include <signal.h>

  int kill(pid_t pid, int sig);
- The **kill** function sends the specified signal, **sig**, to the process whose identifier is given by pid.
- **int raise (int signo)** function sends the signal **sigio** to itself
- Signals provide us with a useful alarm clock facility. The alarm function call can be used by a process to schedule a SIGALRM signal at some time in future.

  #include <unistd.h>

  unsigned int alarm(unsigned int seconds);
- A robust signal interface
  - sigaction() and sigqueue()

An Alarm Clock (alarm.c)

```
static int alarm_fired = 0;
void ding(int sig)
{
    alarm_fired = 1;
}
int main() {
    pid_t pid;
    printf("alarm application
        starting\n");
    pid = fork();
    switch(pid) {
    case -1:    /* Failure */
        perror("fork failed");
        exit(1);
    case 0:     /* child */
        sleep(5);
        kill(getppid(), SIGALRM);
        exit(0);
    }
```

```
/* This is the parent process */
printf("waiting for alarm to go
    off\n");
(void) signal(SIGALRM, ding);
pause();
if (alarm_fired) printf("Ding!\n");
printf("done\n");
exit(0);
}
```

```
$ ./alarm
alarm application starting
waiting for alarm to go off
<5 second pause>
Ding!
done
$
```

A process is a program in execution. It consists of the executing program code, a set of resources such as open files, internal kernel data, an address space, one or more threads of execution and a data section containing global variables.

Fork creates a new process which is a copy of the calling process. That means that it copies the callers memory **(code, globals,heap and stack), registers and file descripters.**

Difference between process and thread:

Process	Thread
Process is an execution of a program. Program byitself is not a process.	Thread is a smallest unit of execution (Light weight process). Thread is a seqence of control with in a process.
Whenever you are creating a process text segment, datasegment, stack segments are created.	Whenever you are creating a thread it will create only stack segment and share text and data segment. i.e., why we are saying thread is smallest unit of execution.
Process is a parallel execution.	Thread is a serial execution.

Drawbacks of Thread

Debugging a multithreaded program is much harder than debugging a single-threaded one, because the interactions between the threads are very hard to control.

Writing multithreaded programs requires very careful design.

A program that splits a large calculation into two and runs the two parts as different threads will not necessarily run more quickly on a single processor machine, as there are only so many CPU cycles to be had, though if nothing else