System calls:

- Function call -> subroutine. System call -> system routine.
- System call arguments passed using registers. Linux/x86 examples
- POSIX API: compiled appropriately to assembly/binary.

User vs Kernel mode:

- determined by a hardware bit.
- User processes run in User mode.
- Call to a system/interrupt routine results in Kernel mode.
- Upon return, system is back in User mode
FAQ: Threads

Processes:
• What happens to a process once it is finished? Resources deallocated, but only after..
• If a process forks a child, what happens to the parent? It continues.
• When the CPU is running user processes how does the OS run?
• Can a process induce its own context switch? Yes, we’ll see how.

Threads:
• A process can have multiple threads
• Consider having multiple program counters per process
  – Multiple locations can execute at once
  • Multiple threads of control -> threads
• Must then have storage for thread details, multiple program counters in PCB. More details later..
Demonstration: Processes

• Mac: apps > utilities > activity monitor > CPU etc.
  • https://support.apple.com/guide/activity-monitor/welcome/mac
    – See information about processes
    – Name, PID, threads, details ..

• Windows 10 Ctrl+Alt+Del
Diagram of Process State

- **Ready to Running**: scheduled by scheduler
- **Running to Ready**: scheduler picks another process, back in ready queue
- **Running to Waiting (Blocked)**: process blocks for input/output
- **Waiting to Ready**: Input available
Meanwhile, on an ordinary Linux kernel...

What's going on with these zombie processes?  

Their parent is too busy to get any notifications...

Daniel Storl {turnoff.us}
Process Control Block (PCB)

Information associated with each process (also called **task control block**)

FAQ: Where is PCB saved?
In memory area that is protected from user access perhaps as structs in a linked list.

<table>
<thead>
<tr>
<th>process state</th>
</tr>
</thead>
<tbody>
<tr>
<td>process number</td>
</tr>
<tr>
<td>program counter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory limits</td>
</tr>
<tr>
<td>list of open files</td>
</tr>
</tbody>
</table>
CPU Switch From Process to Process

Diagram showing the process of switching from process $P_0$ to process $P_1$:

- Process $P_0$ is executing.
- A system call or interrupt occurs.
- The state of $P_0$ is saved into PCB$_0$.
- Process $P_0$ is idle.
- Process $P_1$ is executing.
- Another system call or interrupt occurs.
- The state of $P_1$ is saved into PCB$_1$.
- Process $P_1$ is idle.
- Process $P_0$ is executed again.
- The state of $P_0$ is reloaded from PCB$_0$.
- Process $P_0$ is idle again.

Note: PCB stands for Process Control Block.
Queueing diagram represents queues, resources, flows

Assumes a single CPU. Common until recently
Schedulers

• **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  – Sometimes the only scheduler in a system
  – Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)

• **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  – Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  – The long-term scheduler controls the **degree of multiprogramming**

• Processes can be described as either:
  – **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  – **CPU-bound process** – spends more time doing computations; few very long CPU bursts

• Long-term scheduler strives for good **process mix**
**Addition of Medium Term Scheduling**

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

![Diagram showing the process flow of medium-term scheduling](image-url)
Multitasking in Mobile Systems

• Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
• Due to screen real estate, user interface limits iOS provides for a
  – Single **foreground** process – controlled via user interface
  – Multiple **background** processes – in memory, running, but not on the display, and with limits
    • Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
• Newer iOS supports multitasking better. iOS 14: picture in picture
• Android runs foreground and background, with fewer limits
  – Background process uses a **service** to perform tasks
  – Service can keep running even if background process is suspended
  – Service has no user interface, small memory use.
ARE YOU SURE YOU WANT TO SIT AT THE KIDS TABLE?
Context Switch

• When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
  • **Context** of a process represented in the PCB
  • Context-switch time is overhead; the system does no useful work while switching
    – The more complex the OS and the PCB ➔ the longer the context switch

• Time dependent on hardware support
  – Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once
Processes creation & termination
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes.
- Generally, process identified and managed via a **process identifier (pid)**
A Tree of Processes in Linux
Process Creation (Cont.)

• **Address space**
  – Child duplicate of parent
  – Child has a program loaded into it

• **UNIX examples**
  – `fork()` system call creates new process
  – `exec()` system call used after a `fork()` to replace the child process’ memory space with a new program
Fork ( ) to create a child process

• Fork creates a copy of process
• Return value from fork (): integer
  – When > 0:
    • Running in (original) Parent process
    • return value is pid of new child
  – When = 0:
    • Running in new Child process
  – When < 0:
    • Error! Perhaps exceeds resource constraints. sets errno (a global variable in errno.h)
    • Running in original process

• All of the state of original process duplicated in both Parent and Child! Almost ..
  – Memory, File Descriptors (next topic), etc...
Process Management System Calls

• UNIX fork – system call to create a copy of the current process, and start it running
  – No arguments!

• UNIX exec – system call to change the program being run by the current process. Several variations.

• UNIX wait – system call to wait for a process to finish

• Details: see man pages

Some examples:

```c
pid_t pid = getpid();  /* get current processes PID */;
waitpid(cid, 0, 0);  /* Wait for my child to terminate. */
exit (0);  /* Quit*/
kill(cid, SIGKILL);  /* Kill child*/
```
Unix Process Management

Two processes are running when `fork()` finishes execution!

```c
pid = fork();
if (pid == 0)
  exec(...);
else
  wait(pid);
```
include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t cid;

    /* fork a child process */
    cid = fork();
    if (cid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed\n");
        return 1;
    }
    else if (cid == 0) { /* child process */
        printf("I am the child %d, my PID is %d\n", cid, getpid());
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        printf("I am the parent with PID %d, my parent is %d, my child is %d\n",getpid(), getppid(), cid);
        wait(NULL);

        printf("Child Complete\n");
    }

    return 0;
}
wait/waitpid

- Wait/waitpid( ) allows caller to suspend execution until child’s status is available
- Process status availability
  - Generally after termination
  - Or if process is stopped
- `pid_t waitpid(pid_t pid, int *status, int options);`
- The value of pid can be:
  - 0 wait for any child process with same process group ID (perhaps inherited)
  - > 0 wait for child whose process group ID is equal to the value of pid
  - -1 wait for any child process (equivalent to `wait( )`)
- Status: where status info needs to be saved
Search for man fork( )
http://man7.org/linux/man-pages/man2/fork.2.html

NAME    fork - create a child process
SYNOPSIS    #include <unistd.h>
         pid_t fork(void);
DESCRIPTION    fork() creates a new process by duplicating the calling
   process. The new process is referred to as the child process. ...
   The child process and the parent process run in separate memory spaces...
   The child process is an exact duplicate of the parent process except for the
   following points: ....
RETURN VALUE    On success, the PID of the child process is returned in the
   parent, and 0 is returned in the child. On failure, -1 is returned in the
   parent, no child process is created, and errno is set appropriately.
EXAMPLE    See pipe(2) and wait(2).
...

errno is a global variable in errno.h
Process Group ID

• Process group is a collection of related processes
• Each process has a process group ID
• Process group leader?
  – Process with pid==pgid
• A process group has an associated controlling terminal, usually the user’s keyboard
  – Control-C: sends interrupt signal (SIGINT) to all processes in the process group
  – Control-Z: sends the suspend signal (SIGSTOP) to all processes in the process group

Applies to foreground processes: those interacting With the terminal
A child Inherits parent’s process group ID. Parent or child can change group ID of child by using setpgid.

By default, a Process Group comprises:
• Parent (and further ancestors)
• Siblings
• Children (and further descendants)

A process can only send signals to members of its process group
• Signals are a limited form of inter-process communication used in Unix.
• Signals can be sent using system call
  – `int kill(pid_t pid, int sig);`
Process Termination

• Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  – Returns status data from child to parent (via `wait()`)  
  – Process’ resources are deallocated by operating system

• Parent may terminate the execution of children processes using the `kill()` system call. Some reasons for doing so:
  – Child has exceeded allocated resources
  – Task assigned to child is no longer required
  – The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

```c
kill(child_pid,SIGKILL);
```
Process Termination

• Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  – **cascading termination.** All children, grandchildren, etc. are terminated.
  – The termination is initiated by the operating system.
• The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process
  \[ \text{pid} = \text{wait}(&\text{status}); \]
• If no parent waiting (did not invoke `wait()`) process is a **zombie**
• If parent terminated without invoking `wait`, process is an orphan (it is still running, reclaimed by init)

Zombie: a process that has completed execution (via the exit system call) but still has an entry in the process table
Multi-process Program Ex – Chrome Browser

• Early web browsers ran as single process
  – If one web site causes trouble, entire browser can hang or crash

• Google Chrome Browser is multiprocess with 3 different types of processes:
  – **Browser** process manages user interface, disk and network I/O
  – **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    • Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  – **Plug-in** process for each type of plug-in

![Chrome Browser](image-url)
Multitasking
Cooperating Processes

• **Independent** process cannot affect or be affected by the execution of another process

• **Cooperating** process can affect or be affected by the execution of another process

• Advantages of process cooperation
  – Information sharing
  – Computation speed-up
  – Modularity
  – Convenience
Interprocess Communication

- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need *interprocess communication* (IPC)
- Two models of IPC
  - Shared memory
  - Message passing
Communications Models

(a) Message passing.  (b) shared memory.
Producer-Consumer Problem

• Common paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  – *unbounded-buffer* places no practical limit on the size of the buffer
  – *bounded-buffer* assumes that there is a fixed buffer size
Bounded-Buffer – Shared-Memory Solution

- **Shared data**
  ```c
  #define BUFFER_SIZE 10
  typedef struct {
    . . .
  } item;
  
  item buffer[BUFFER_SIZE];
  int in = 0;
  int out = 0;
  ```

- **in** points to the **next free position** in the buffer
- **out** points to the **first full position** in the buffer.
- Buffer is empty when **in == out**;
- Buffer is full when 
  
  $$(\text{in} + 1) \mod \text{BUFFER SIZE} == \text{out}. \quad \text{(Circular buffer)}$$

- This scheme can only use **BUFFER_SIZE-1** elements

![Circular buffer diagram]

(2+1)%8 = 3 but (7+1)%8 = 0
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}

Out

In

0 1 2 3 4 5 6 7
Producer Consumer Mismatch
FAQ

• Can multiple processes/threads run on a single CPU? Yes, “concurrently” but not in “parallel” without additional hardware.

• Why fork? The only way to create new processes, except for init.

• BSD (Berkeley Software Distribution) is a derivative of original Unix.

• Linux is an implementation of Unix. Clone. “Unix-like”

• Process Scheduling – How? more soon