Processes, System calls, & POSIX
Processes
Program

• List of instructions and data
• When running a program, OS asks CPU to execute the program.
Process

- Running program
  - E.g., web browser and mail program

- OS virtualizes CPU
  - Time sharing mechanism (context switching)
    - Stop one process and starts another process
  - Scheduling policy
    - Decide what program runs on a given CPU
    - Consider metrics e.g., historical information
Process (cont.)

• Process is an abstraction of a running program.

• Process consists of
  – Memory (address space)
  – Registers e.g., program counter (PC)
  – I/O devices
Process Creation

• Load a program code and static data into an address space
  ▪ Program is executable
• Allocate memory (stack) for program’s
  ▪ Local variables, function parameters, return addresses
  ▪ argc and argv (the main method)
• Allocate memory (heap)
  ▪ Dynamically allocated memory through e.g. malloc
• I/O related tasks
Process API
Process API

• Process-related calls that programs make:
  • fork( )
    – Creates almost an exact copy of the calling process
  • wait( )
    – Parent process waits for child process to complete its execution.
  • exec( )
    – Runs a program in place of the copy of the original program
fork

• Compile and run p1.c in this directory:

/home/hajiamini/courses/CSC213/ostep-code/p1.c

• Process identifier (PID)
• `fork()` creates a process (child) that is identical to the calling process (parent).
• After fork, parent and child go separate ways.
wait

• Compile and run p1.c in this directory:
  /home/hajiamini/courses/CSC213/ostep-code/p2.c

• wait() waits for a child process to finish.
• Child finishes earlier than parent.
exec

• Compile and run p1.c in this directory:

/home/hajiamini/courses/CSC213/ostep-code/p3.c

• exec() runs a different program than the original program.

• Program code and data of the current program is replaced by the new program.
Limited Direct Execution
Direct Execution

- Run a program on CPU without restrictions

**Execution Time Line**

**OS**
- Create an entry for process list
- Allocate memory
- Load program code into memory
- Set up stack with argc/argv
- Execute Call main()

**Program**
- Run main()
- Execute return from main

Free memory allocated to process
Remove entry from process list
Problem

• Program performs restricted operations (e.g., access to disk or use more resources)
  – Solution: two modes of operation
    ▪ User mode: user program runs
    ▪ User mode: OS runs
  – Process executes restricted (privileged) requests through system calls.
    • Systems calls are C procedure calls e.g., malloc(), open()
Problem (cont.)

• Execute system calls via special instructions
  – trap: go from user mode to kernel mode
  – Return-from-trap: go back from kernel mode to user mode

• Trap instructions are hand-coded inside C procedure calls.
Limited Direct Execution Protocol

<table>
<thead>
<tr>
<th>OS @ boot (kernel mode)</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize trap table</td>
<td>remember address of syscall handler</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS @ run (kernel mode)</th>
<th>Hardware</th>
<th>Program (user mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create entry for process list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocate memory for program</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load program into memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Setup user stack with argv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fill kernel stack with reg/PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return-from-trap</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Handle trap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do work of syscall</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return-from-trap</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free memory of process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove from process list</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
POSIX Presentations
POSIX

• Application interface used by UNIX-like operating systems

• System calls look like library function calls in C.
  – System calls are implemented in the assembly language.
  – A system call executes a trap instruction.
POSIX (cont.)

- **sscanf**
  - Reads from the string `s`

```c
#include <stdio.h>
#include <string.h>

int main( ){
    char first_name[20];
    char last_name[20];
    int height;
    float weight;
    sscanf( "jerod weinmann 178 77.32", "%s %s %d %f", first_name, last_name,
            &height, &weight );
    printf( "%s %s is a professor who is %dcm tall and weighs %.2fkh.\n", first_name, last_name, height, weight ) ;
    return 0;
}
```
POSIX (cont.)

• **strtod**
  
  — Converts string to double

```c
#include <stdio.h>
int main( ){
    char string[50] = "25.125 Hello";
    char* toDouble;
    double doub;

    doub = strtod( string, &toDouble );
    printf( "Double = %f\n", doub );
    printf( "String = %s\n", toDouble )
    return 0;
}
```
POSIX (cont.)

• `strcmp`
  – Compares two strings

```c
int diff;
diff = strcmp( "aa", "aa" );       // diff = 0

diff = strcmp( "aea", "aba" );     // diff = 1
```
POSIX (cont.)

• `strchr`
  
  — Finds the first occurrence of a character in a string

```c
#include <stdio.h>
#include <string.h>
int main( ){
    char string[ ] = "This is a string for testing";
    char* p;
    p = strchr( string, 'i' );

    printf( "Character i is found at position %d\n", p - string + 1 );

    return 0;
}
```
POSIX (cont.)

- **strsep**
  - Finds the first token separated by a delimiter

```c
#include <stdio.h>
#include <string.h>

int main( ){
    char ch1  = "This is a test";
    char* ch1Pointer = ch1;
    char* ch1Removed = strsep( &ch1Pointer, " ");
    printf( "%s\n", ch1Pointer );
    printf( "it returns: %s\n", ch1Removed );

    return 0;
}
```
POSIX (cont.)

• memmove
  – Copies a specified amount of bytes from a memory area to another memory area

```c
#include <stdio.h>
#include <string.h>
int main( ){
    char* str1 = "Hello World";
    char* str2 = malloc( sizeof( char ) * 11 );
    printf( "Before: \n String1 = %s\n String2 = %s\n", str1, str2 );
    memmove( str2, str1, 11 );
    printf( "After: \n String1 = %s\n String2 = %s\n", str1, str2 );
    return 0;
}
```
• `snprintf`
  — Redirects output to a string buffer

```c
#include <stdio.h>
int main() {
    char buffer[10];
    char* s = "Hello World!";
    int j = snprintf(buffer, 6, "%s\n", s);
    printf("%d, %s\n", j, buffer);
    return 0;
}
```
POSIX (cont.)

• perror
  – Prints a descriptive error message

```c
#include <stdio.h>
int main( ){
    FILE* fptr = fopen("/home/hajiamini/passwd", "r");
    if(fptr == NULL){
        perror("fopen");
    }
    return 0;
}
```
POSIX (cont.)

- **strstr**
  
  — Finds a substring within a string

```c
#include <stdio.h>
int main( ){
    const char haystack[20] = "Hello World!";
    const char needle[20] = "World!";
    char* substr;
    substr = strstr(haystack, needle);
    return 0;
}
```
• **getline**
  
  — Copies a line, read from file, to a string

```c
#include <stdio.h>

int main( ){
  FILE* stream;
  char* line = NULL;
  size_t len = 0;
  stream = fopen(argv[1], “r”);
  getline(&line, &len, stream);
  printf(“%s”, line);
  free(line);
  fclose(stream);
  return 0;
}
```
POSIX (cont.)

• gets
  — Copies a line, read from console, to a string

#include <stdio.h>
int main( ){
    char str[20];
    gets(str);
    printf("%s\n", str);
    return 0;
}