Operating Systems

Christopher Kruegel
Department of Computer Science
UC Santa Barbara
http://www.cs.ucsb.edu/~chris/
The Process Concept

• The OS creates number of virtual computers
• Execution of a program on one of these virtual computer is called a sequential process
• The virtual computer gives the illusion to each process that it is running on a dedicated CPU with a dedicated memory
• The actual CPU is switched back and forth among the processes (multiprogramming with time-sharing)
• Process memory is managed so that all the needed portions are present in the actual memory
• The virtual computer is the execution environment, the process is the executor, and the program being executed determines the process behavior
Programs and Processes

- Static object existing in a file
- A sequence of instructions
- Static existence in space & time
- Same program can be executed by different processes

- Dynamic object – program in execution
- A sequence of instructions executions
- Exists in limited span of time
- Same process may execute different program

```c
main() {
    int i, prod = 1;
    for (i=0 ; i < 100; i++)
        prod = prod * i;
}
```

`prod = prod*i;`  
Process executes it 100 times
Process Life Cycle

- A process can be created
  - During OS initialization
    • “init” process in UNIX
  - By another process
    • fork(), or NtCreateProcess()

- A process can be terminated
  - By itself
    • exit(), or ExitProcess()
  - Because of an error
    • e.g., segmentation fault
  - By another process
    • kill(), TerminateProcess()
Process States

- Process states
  - Running (using the CPU)
  - Ready (waiting for the CPU)
  - Blocked (waiting for a resource to become available)
Process States

• Process hierarchy
  – each process has a parent
  – each process can have many children
  – does not have to be like that (e.g., Windows NT)

• Parent must collect status of child processes
  – otherwise, children become zombie processes
  – what happens when parent dies first?

• How is signal delivery handled
  – do children receive signals of parents?
Process Implementation

- The OS maintains a *process table* with an entry for each process, called *Process Control Block* (PCB)

- The PCB contains
  - Process ID, User ID, Group ID
  - Process state (Running, Ready, Blocked)
  - Registers (Program counter, PSW, Stack pointer, etc)
  - Pointers to memory segments (Stack, Heap, Data, Text)
  - Priority/Scheduling parameters
  - Accounting information
  - Signal management functions
  - Open file tables
  - Working directory
Threads

- A process is a way to
  - Group resources (memory, open files, ...)
  - Perform the execution of a program: a thread of execution (code, program counter, registers, stack)

- Multiple threads of execution can run in the same process environment

- Multiple threads share
  - Common address space (shared memory)
  - Open files
  - Process, user, and group IDs

- Each thread has its own code, program counter, set of registers, and stack
Threads

(a)

(b)
1: int i;
2: 
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7: 
8: f()
9: {
10:     g();
11: }
12: 
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```c
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```
Parallel Processes

1: int i;
2: 
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7: 
8: f()
9: {
10:     g();
11: }
12: 
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }

Address space (Data)

| i = 42 |
| P1 |

| i = ? |
| P2 |

Registers (here: Program Counter)

| PC = 10 |
| P1 |

| PC = 15 |
| P2 |

Stack

| 17 |
| P1 |

Running

Context
Parallel Processes

1: int i;
2: 
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7: 
8: f()
9: {
10:    g();
11: }
12: 
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
Parallel Processes

```
1: int i;
2: 
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7: 
8: f()
9: {
10:     g();
11: }
12: 
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```

**Address space (Data)**

- i = 42
- i = 17

**Registers (here: Program Counter)**

- PC = 10

**Stack**

- 17

**Running Context**
Parallel Processes

```c
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```

**Address space (Data)**
- `i = 42`
  - P1
- `i = 17`
  - P2

**Registers (here: Program Counter)**
- `PC = 10`
  - P1
- `PC = 5`
  - P2

**Stack**
- `17`
  - P1
- `17
  - 11`
  - P2

**Running Context**
1: int i;
2: 
3: g() 
4: { 
5:     printf("Value of i is %d\n", i);
6: } 
7: 
8: f() 
9: { 
10:     g();
11: } 
12: 
13: int main(int argc, char **argv) 
14: { 
15:     i = get_input();
16:     f();
17:     return 0;
18: }

Value of i is 17
Parallel Processes

1: int i;
2:
3: g()
4: {
5:   printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:   g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:   i = get_input();
16:   f();
17:   return 0;
18: }

Address space (Data)
- i = 42
- i = 17

Registers (here: Program Counter)
- PC = 5
- PC = 6

Stack
- 17
- 17
- 11
- 11

Running
- P1
- P2

Context
- P1
- P2
Parallel Processes

1: int i;
2: 
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7: 
8: f()
9: {
10:     g();
11: }
12: 
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }

Value of i is 42

Address space (Data)

i = 42
PC = 6

Registers (here: Program Counter)

i = 17
PC = 6

Stack

17
11

Running

P1
P2

Context

P1
P2
1: int i;
2: 
3: g()
4: {
5:    printf(“Value of i is %d\n”, i);
6: }
7: 
8: f()
9: {
10:     g();
11: }
12: 
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }

Address space (Data)

Registers (here: Program Counter)

Stack

Running

Context
1: int i;
2: 
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7: 
8: f()
9: {
10:     g();
11: }
12: 
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }

Address space (Data)
i = 42

Registers (here: Program Counter)
PC = 16
PC = 15

Stack
T1
T2

Running
T1
T2

Context
1: int i;
2: 
3: g()
4: {
5:    printf("Value of i is %d\n", i);
6: }
7: 
8: f()
9: {
10:   g();
11: }
12: 
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }

---

**Address space (Data)**

- `i = 42`

**Registers (here: Program Counter)**

- `PC = 10`
  - T1
- `PC = 15`
  - T2

**Stack**

- `17`
  - T1
  - T2

**Running**

- T1
- T2

**Context**
int i;
g();
{
    printf("Value of i is %d\n", i);
}
f();

int main(int argc, char **argv)
{
    i = get_input();
f();
    return 0;
}
```c
1: int i;
2: 
3: g()
4: {
5:   printf("Value of i is %d\n", i);
6: }
7: 
8: f()
9: {
10:   g();
11: }
12: 
13: int main(int argc, char **argv)
14: {
15:   i = get_input();
16:   f();
17:   return 0;
18: }
```
```c
1: int i;
2: 
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7: 
8: f()
9: {
10:     g();
11: }
12: 
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```
```c
1: int i;
2:
3: g()
4: {
5:    printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

**Value of i is 17**

### Address space (Data)

- `i = 17`

### Registers (here: Program Counter)

- `PC = 10`
- `PC = 6`

### Stack

- `17`
- `17`
- `11`

### Running

- T1
- T2
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }

Address space (Data)

Registers (here: Program Counter)

Stack

Running

Context
int i;
g()
{    printf("Value of i is %d\n", i);
}
f()
{
    g();
}
int main(int argc, char **argv)
{
    i = get_input();
f();
    return 0;
}
Why Threads?

• Useful to structure applications that have to do many things concurrently
  – One thread is waiting for I/O
  – Another thread *in the same process* is doing some computation

• Having threads share common address space makes it easier to coordinate activities

• Use a shared data-structure through which the processes can be coordinated:
  – Producer-Consumer interactions
  – Shared data structures/counts

• More efficient than using processes (context switch is faster)
Thread Primitives

- thread_create
- thread_exit
- thread_join
- thread_yield

(synchronization primitives)
Thread Implementation

• Threads can be implemented in user space
  
  – Pros
    • Performance (no kernel/user switch)
    • Portability (same primitives for every environment)
    • Flexibility (custom scheduling algorithm)
  
  – Cons
    • Blocking system calls block the process, not the thread
      – need to check if a system call would block before each invocation
    • Threads cannot be easily preempted (they have to yield)
Thread Implementation

• Threads can be implemented in the kernel

  – Pros
    • Blocking system calls suspend the calling thread only
    • Can take advantage of multiple CPUs
    • Signals can be delivered more precisely

  – Cons
    • Can be heavy, not as flexible
Threading Issues

- What happens on a fork()?
  - only a single thread is created in the child

- What happens with shared data structures and files?
  - threads need to be careful and synchronize access

- What about stack management?
  - each thread needs its own stack

- What about signal delivery?
  - complicated!
  - some signals are sent to specific thread (alarm, segfault)
  - others to the first that does not block them (termination request)
Reentrant Functions

• What about global variables in libraries?
  – functions need to be reentrant

• Some functions are not designed to be invoked concurrently
  – Use of global variables, such as errno

• Functions used by threads need to be reentrant
Portability Issues and Pthreads

- POSIX 1003.1c (a.k.a. pthreads) is an API for multi-threaded programming standardized by IEEE as part of the POSIX standards.

- Most Unix vendors have endorsed the POSIX 1003.1c standard.

- Implementations of 1003.1c API are available for many UNIX systems.

- pthreads defines an interface
  - implementation can be done in either user or kernel space.

- Thus, multithreaded programs using the 1003.1c API are likely to run unchanged on a wide variety of Unix platforms.