Operating Systems

VI. Threads

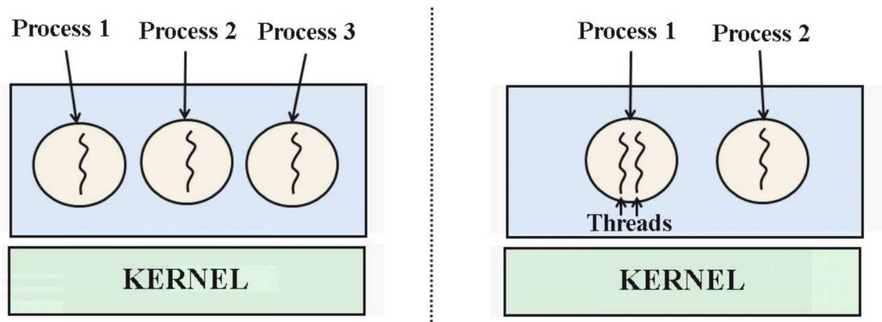
Ludovic Apvrille
ludovic.apvrille@telecom-paris.fr
Eurecom, office 470

`perso.telecom-paris.fr/apvrille/OS/`

Processes: Are they a Good Idea?

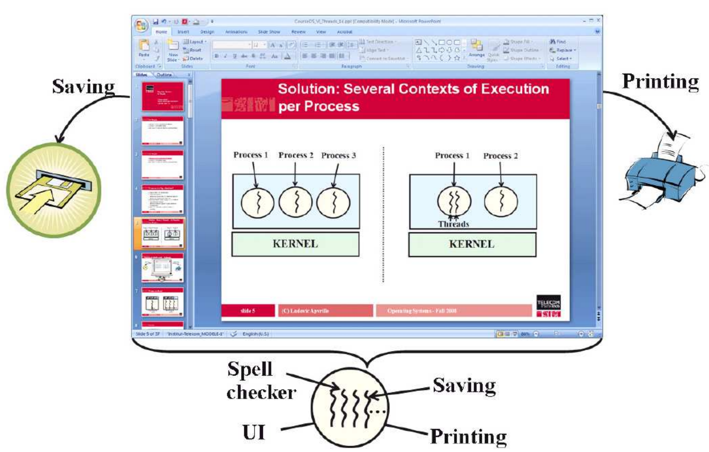
- **Long creation / destruction time**
- **Long switching time**
  - Frequent process switching on interactive systems
  - Switching of processes takes time
- **Communications between processes is cumbersome**
  - Because of protection reason!
  - Need of explicit communication mechanisms
    - Shared memory, etc.
- **Processes consume a lot of resources**
  - e.g., memory

Solution: Several Contexts of Execution Per Process



Processes | Processes + threads

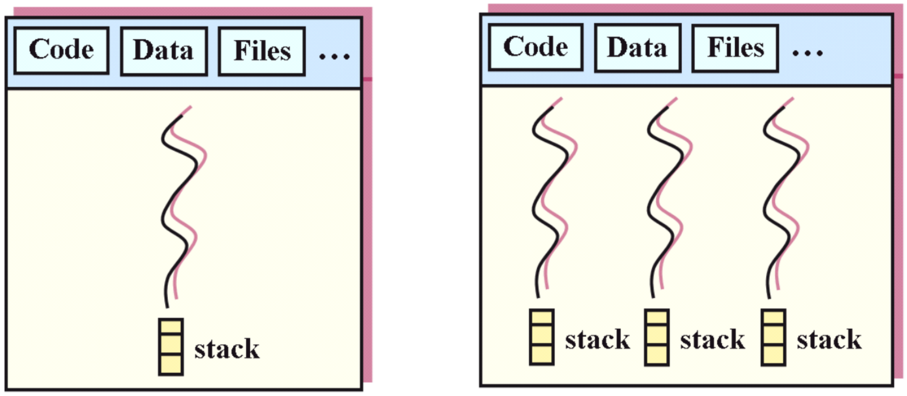
Example of a Multithreaded Application



Do you know such other applications?

(BTW: I do not make my slides with Power Point → use LaTeX- Beamer!)

Processes vs. Threads

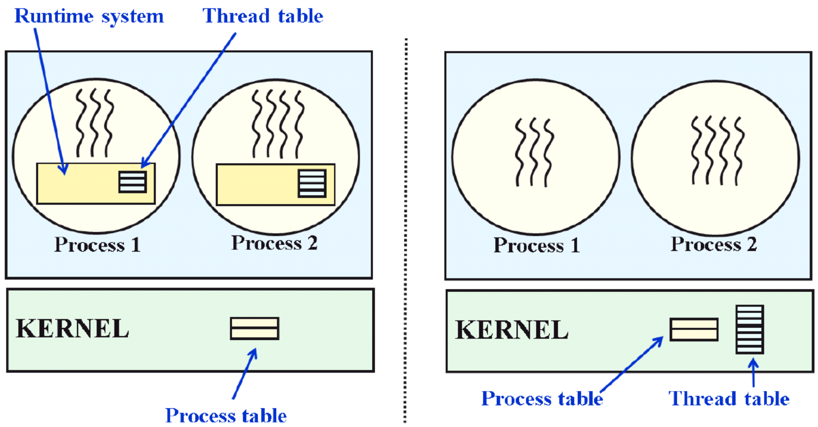


Mono-thread process Multi-thread process

Benefits

- **Responsiveness**
  - An interactive application can continue to execute even if one of its activity is blocked
    - Web browser: page scrolling when images are being loaded
- **Resource sharing**
  - Several threads share several resources, e.g.,
    - Part of their process address space
    - Open files, open network connections, ...
  - But no memory protection among threads
- **Performance**
  - Threads Creation/Switching/Destruction is much faster than for processes
- **Utilization of multi-core & multi-thread architectures**
  - Each thread of a process may be running in parallel on a different processor
  - Each thread of a process may be running in parallel on the same processor core supporting hyper-threading technologies

User and Kernel threads



→ Different multithreading models

Indeed, most OS support both user and kernel threads

User and Kernel threads (Cont.)

User threads: implemented by a thread library at user level

- All management is done at user level
  - Kernel is not aware of threads

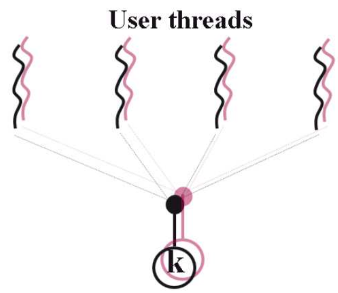
Kernel threads: threads are supported directly by the OS

- Creation, management, scheduling is done by kernel
- Windows NT, Solaris, Linux, MacOS, Android, iOS: kernel threads

Multithreading Models

Many-to-One Model

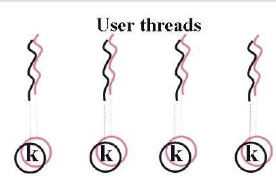
- Thread management done at user level
  - Efficient
  - Scheduling can be customized
- But: a blocking system call performed by one of the threads of a process implies that all threads are blocked



Multithreading Models (Cont.)

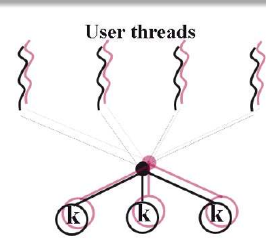
One-to-One Model

- Best concurrency
- Performance drawbacks: creation of threads, etc.



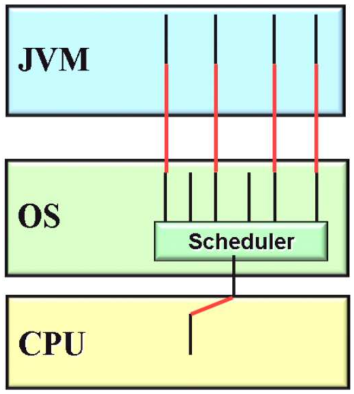
Many-to-Many Model

- Trade-off between performance and degree of concurrency
- Solaris



Java (Native) Threads

- Java threads are mapped onto kernel threads
- **But**: Scheduling differs according to target platform
  - "Green threads" managed only by the JVM



Dalvik Threads (Android)

- Dalvik threads are attached to kernel threads (in fact, to *pthreads*, just like in Linux)
- States of threads, see "public static final enum Thread.State"

Enum values	
Thread.State	BLOCKED Thread state for a thread blocked waiting for a monitor lock.
Thread.State	NEW Thread state for a thread which has not yet started.
Thread.State	RUNNABLE Thread state for a runnable thread.
Thread.State	TERMINATED Thread state for a terminated thread.
Thread.State	TIMED_WAITING Thread state for a waiting thread with a specified waiting time.
Thread.State	WAITING Thread state for a waiting thread.

Linux Threads

- At first, each thread was implemented as a process with memory sharing features
- Since kernel 2.6: One-to-one multithreading model
  - NPTL for Linux - Native POSIX Thread Library for Linux
  - Kernel scheduler doing all of the scheduling of threads
  - Better performance, in particular for server systems (database systems, etc.)
- To know the default threading model of your system:

```
$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.24
```


POSIX Threads

- Manipulation of threads from C programs
- At compilation, you must use the -lpthread directive

Creating and joining threads

`pthread_create() pthread_join()`

Terminating threads

`pthread_cancel() pthread_exit() exit() return();`

Synchronizing threads

`...(attend next lectures!)`

Basics

- Processes / threads need to exchange data
  - Processes/ threads must have a way to refer to each other
  - Threads share the same address space
- Types of communication:
  - Direct communication
  - Indirect communication

A few inter-process/inter-thread communications of Linux:

Signals, pipes, message passing, semaphores, shared memory, network sockets, etc.

Pipes

- One-way data stream communication
- Communication routed by kernel

Shell command: |

```
$ vmstat -s | grep fork
```

instead of:

```
$ vmstat -s > temp
$ grep fork temp
```

Signals

Signals = software interrupts

- Asynchronous events
- For example, signal to stop a process (CTRL-C)
- The OS forwards signals to the destination process

Signals under Linux

- 31 signals
- Name = SIG*: SIGKILL, SIGSTOP, SIGTRAP, etc.
- Three ways to manage signals that are received
  1. Ignore signals (except SIGKILL, SIGSTOP)
  2. Catch signals
  3. Let the default action apply
- If the signal is not ignored, the process is awoken (if necessary)

UNIX System Calls for Sending and Receiving Signals

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

pid of the destination process

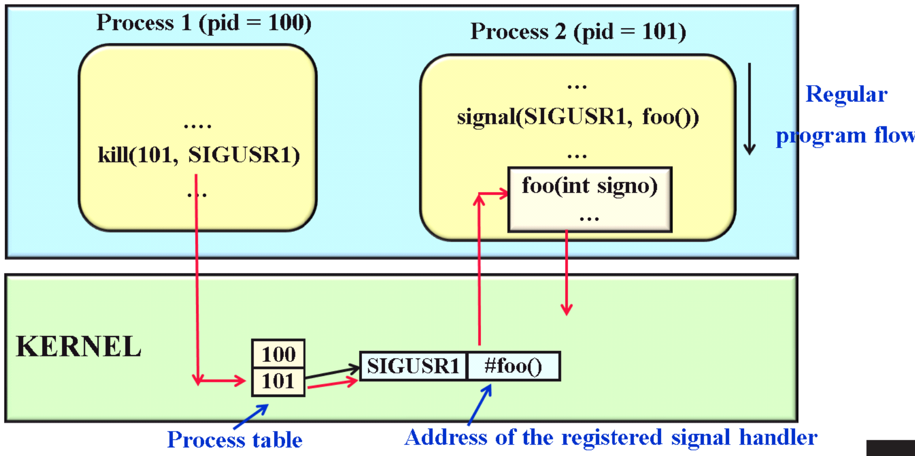
Number of the signal (9=SIGKILL, etc.)

(range 1->31)

```
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

Function called when the signal #signum is received

Sending and Catching a Signal



Code at Receiver's Side

```
#include <signal.h>

void getSignal(int signo) {
  if (signo == SIGUSR1) {
    printf("Received SIGUSR1\n");
  } else {
    printf("Received%d\n", signo);
  }
  return;
}

int main(void) {
  printf("Registering #SIGUSR1=%d\n", SIGUSR1);
  signal(SIGUSR1, getSignal);
  sleep(30);
  printf("End of sleep\n");
}
```

(Simplified code... ALWAYS test the return value of all functions!)

Code at Sender's Side

```c
#include <sys/types.h>
#include <signal.h>

int main(int argc, char**argv) {
  int pid;

  if (argc <2) {
    printf("Usage: sender <destination process pid>\n");
    exit(-1);
  }

  pid = atoi(argv[1]);
  printf("Sending SIGURG to %d\n", pid);
  if (kill(pid, SIGURG) == -1) return;
  printf("Sending SIGUSR1 to %d\n", pid);
  if (kill(pid, SIGUSR1) == -1) return;
}
```


Let's Execute this Code!

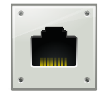
Shell 1

```
$ gcc -o receiver receiver.c
$ receiver
Registering #SIGUSR1=10







Received SIGUSR1
End of sleep
$
```

Shell 2

```
$ gcc -o sender sender.c


$ ps
 PID TTY          TIME CMD
23930 pts/1    00:00:00 bash
2241 pts/1     00:00:00 receiver
2242 pts/1     00:00:00 ps
$ sender 2241
Sending SIGURG to 2241
Sending SIGUSR1 to 2241
$
```

Network Sockets



Definition

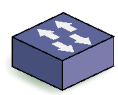
Bidirectional communication point with an associated address and a communication protocol

- Address: IP address, port number
- Prococol: TCP, UDP
- Can be used for local or remote communication

Socket types

- TCP sockets are also called *stream sockets*
  - Connection-oriented communication
- UDP sockets are also called *datagram sockets*
  - Connectionless communication

TCP Sockets



Server

```
int serverfd, serverconn;
struct sockaddr_in s_addr;
char sendBuff[1024];
...
int serverfd=socket(AF_INET,SOCK_STREAM,0);
s_addr.sin_family = AF_INET;
s_addr.sin_addr.s_addr=htonl(INADDR_ANY);
s_addr.sin_port = htons(1234);
bind(serverfd, (struct sockaddr*)&s_addr,
  sizeof(s_addr));
listen(serverfd, 10);
..

serverconn = accept(serverfd, (struct
  sockaddr*)NULL, NULL);
...
//Fill buffer
...
write(serverconn, sendBuff,
  strlen(sendBuff));
```

Client

```
int clientfd = 0, n = 0;
char recvBuff[1024];
struct sockaddr_in s_addr;
memset(recvBuff, '0',sizeof(recvBuff));

clientfd = socket(AF_INET,SOCK_STREAM,0));

inet_pton(AF_INET, "<IP of Server>",
  &s_addr.sin_addr)
s_addr.sin_family = AF_INET;
s_addr.sin_port = htons(1234);

connect(clientfd,
(struct sockaddr *)&s_addr,
  sizeof(s_addr))

while ((n = read(clientfd, recvBuff,
  sizeof(recvBuff)-1)) > 0) {
  ...
}
```

(Simplified code... ALWAYS test the return value of all functions!)

IPC System V

Basics

- Each object is referred in the kernel by a non-negative integer (identifier)
- Objects remain in the kernel until an explicit delete command is executed
- Can be used by multiple processes

Objects

- Shared memory segments, semaphores, message queues

Shell Commands

- *ipcs*: Prints information about IPC objects
- *ipcrm*: Removes one or more IPC objects

Example of Shell Commands

$ ipcs

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems      status
0x00000000 98307      apvrille  600        1
0x00000000 131076     apvrille  600        1
0x00000000 163845     apvrille  600        1
0x00000000 622611     apvrille  666        1
0x00000000 655380     apvrille  666        1

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

Example of Shell Commands (Cont.)

$ ipcsrm sem 98307 131076 163845 622611 655380

$ ipcs

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems      status

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```


Shared Memory

Basics

- Two or more processes may share a given region of memory
- Most efficient way to exchange data because there is no copy of data between one process address space to another

Creating a segment of shared memory

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int flag);
```

Shared Memory (Cont.)

Controlling created segments of shared memory:

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

A process can attach a shared memory segment to its address space:

```
void* shmat(int shmid, void *addr, int flag);
```

A process can detach a shared memory segment from its address space:

```
void* shmdet(void *addr);
```