Overview

Introduction

Distributed Programming

Distributed programming is about processes which communicate over a network

- Obvious requirement: good knowledge in how to handle processes locally

Processes

Basic idea

We build **virtual processors** in software, on top of physical processors:

- **Processor:** Provides a set of instructions along with the capability of automatically executing a series of those instructions.
- **Thread:** A minimal software processor in whose • a series of instructions can be executed. Saving a thread context implies stopping the current execution and saving all the data needed to continue the execution at a later stage.
- **Process:** A software processor in whose context one or more threads may be executed. Executing a thread, means executing a series of instructions in the context of that thread.

Threads and Distributed Systems

Multithreaded Web client

Hiding network latencies:

- Web browser scans an incoming HTML page, and finds that **more files need to be fetched**
- **Each file is fetched by a separate thread**, each doing a (blocking) HTTP request.
- As files come in, the browser displays them.

Multiple request-response calls to other machines (RPC)

Hiding network latencies:

- A client does several calls at the same time, each one by a different thread.
- It then waits until all results have been returned.
- Note: if calls are to different servers, we may have **a linear speed-up**.

Processes

- One process is made of:
  - A process identifier (an integer)
  - One executing program
  - Memory used to execute the program
  - One program counter (indicates where in the program the process currently is)
  - A number of signal handlers (tells the program what to do when receiving signals)
- One process can determine what is its own identifier:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

- It can also determine the identifier of its parent:

```
pid_t getppid(void);
```

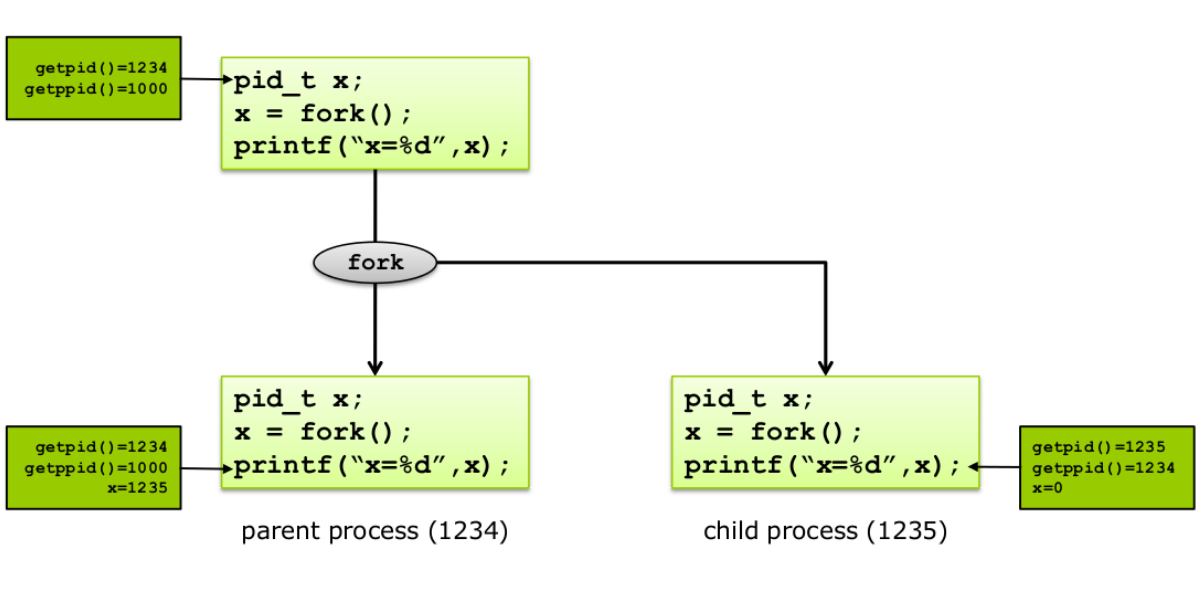
Processes

- Computers can execute programs
- A process is one instance of a program while it is executing
  - For example: I am currently using acroread to display these slides
  - At the same time, there are many other programs executing: they are other processes
- The same program can be executed multiple times in parallel
  - Somebody else may log on my computer and start acroread
  - These are several separate processes, executing the same program
- A process can only be created by another process
  - E.g., when I type a command, my shell process will create an acroread process

The fork() Primitive

- There is exactly one way of creating a process in Unix:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```



The fork() Primitive

- The child process is an exact copy of the parent
  - It is running the same program
  - Its program counter is at the same position within the program
    I.e., just after the fork() call
  - Its memory area is an exact copy of the parent's memory
  - Signal handlers and file descriptors are copied too
- There is one way of distinguishing between the two processes:

The fork() Primitive

- There is one way of distingusing between the two processes:
  - *fork*() return 0 to the child process
  - *fork*() returns the childs pid to the parent ($pid > 0$)
  - *fork*() return -1 if an error occurred
- Most often, programs need to check the return value from *fork*()

The fork() Primitive

- After testing for *fork*() 's return value, the two processes may diverge:

```
    pit_t pid;
    pid = fork();
    if (pid<0)
    {
    perror("Fork error"); exit(1);
    }
 if (pid==0)
    {
    printf("I am the child process\n");
    while (1) putchar("c");
 } else {
          printf("I am the parent process\n");
      while (1) putchar("p");
     }
```

The Fork of Death

- You must be careful when using *fork*()!
  - It is very easy to create very harmful programs
- What does the following program do?

```
while (1)
{
  fork();
}
```

The exec() Primitives

- With *fork*(), children always execute the same program as their parent
- *exec*() allows one process to switch to another program
  - All code/data for that process are destroyed
  - Environment variables and file descriptors are kept identical
  - New code is loaded, then started (from the beginning)

The exec() Primitives

- There are 4 versions of the exec() primitive:
  - Depending if the program is searched in $*PATH* or if it is specified by an absolute path
  - Depending if program parameters are passed as parameters to *exec*() or as a single array:

	absolute path.	**search in** $*PATH*
n parameters	*execl*()	*execlp*()
1 array	*execl*()	*execlp*()

- *execl*()*example* :

```
execl("/bin/ls", "ls", "-l", "/home/wondimagegn", NULL);
```

- *execv*()*example* :

```
char *params[4] = {"ls", "-l", "/home/wondimagegn", NULL};
execv("/bin/ls", params);
```

Stopping a Process

- A process stops when:
  - The main() function returns or
  - The program calls exit()
- A process cannot directly kill another process
  - It can only send a signal to it
  - Depending on the destination's signal handlers, it may stop (or do something else)
  - By default, the signal SIGINT stops the process.This is what happens when you type ^{C}

Signal Handling

- Signals can be sent by the system to a process
  - SIGSEGV: segmentation fault (non-authorized memory access)
  - SIGBUS: bus error (non-aligned memory access)
  - SIGPIPE: you tried to write in a pipe with no reader
  - SIGCHLD: one of your children process has stopped
  - There are two generic signals to be used by user programs: SIGUSR1 and SIGUSR2
  - You can get a complete list of signals with kill -l

Signal Handling

- To setup a signal handler:

```
#include <signal.h>
void (*signal(int signum, void (*sighandler)(int)))(int);
```

- Example:

```
#include <signal.h>
void myhandler(int sig) {
printf("I received signal number %d!\n",sig);
signal(sig,myhandler);
}
int main () {
void (*oldhandler)(int);
oldhandler = signal(SIGINT,myhandler);
signal(SIGUSR1,myhandler);
...
}
```

Inter-Process Communication

- In general, processes cannot influence each other
  - Each process is executed in isolation from the others
  - For example, one process cannot write into the memory of another process. . .
- But you can decide to make several processes communicate together:
  - Send a signal to another process
  - Pipes: a communication channel to transfer data
  - Shared memory: the same memory area is accessible to multiple processes
  - Semaphores: perform synchronization (e.g., to regulate access to shared memory)
- All these IPCs only work between processes of the same computer!

Pipes

- A pipe is a unidirectional communication channel between processes
  - All data written into it can be read at the other end
  - If you need bidirectional communication, use two separate pipes!
- To create a pipe: pipe()

```
#include <unistd.h>
int pipe(int fd[2]);
```

Pipes

- Pipes are often used in combination with *fork*()

```
 int main() {
int pid, fd[2];
char buf[64];

if (pipe(fd)<0) exit(1);
pid = fork();
if (pid==0)        /* child */
{
  close(fd[0]);
  /* close reader */
  write(fd[1],"hello, world!",14);
}
else  /* parent */
{
  close(fd[1]);/* close writer */
  if (read(fd[0],buf,64) > 0)
  printf("Received: %s\n", buf);
  waitpid(pid,NULL,0);
}
}
```

Pipes

- Pipes are used for example for shell commands like:

```
#include <unistd.h>

sort foo.txt | uniq | wc
```

- Pipes can only link processes which have a common ancestor
  - Because children processes inherit the file descriptors from their parent

Shared Memory

- Shared memory allows multiple processes to have direct access to the same memory area
  - They can interact through the shared memory segment
- Shared memory segments must be created, then attached to a process to be usable. Then, you must detach and destroy it afterwards.
- When using shared memory, you must be very careful about race conditions, and solve them thanks to semaphores.

Shared Memory

- To destroy a segment:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- Shared memory segments stay persistent even after all processes have died!
  - You must destroy them in your programs
  - The ipcs command shows existing segments (and semaphores)
  - You can destroy them by hand with: *ipcrmshm* < *id* >

Shared Memory

- To create a shared memory segment:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

- **key:** rendezvous point key=IPC_PRIVATE if it will be used by children processes
- **size:** size of the segment in bytes
- **shmflg:** options (access control mask)
- **Return value:** a shm identifier (or -1 for error)

Shared Memory

- To **attach** a shared memory segment:

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

  - **shmid:** shared memory identifier (returned by shmget)
  - **shmaddr:** address where to attach the segment, or NULL if you don't care
  - **shmflg:** options (access control mask)

- To **detach** a shared memory segment:

```
int shmdt(const void *shmaddr);
```

  - **shmaddr:** segment address
  - **Attention: shmdt() does not destroy the segment!**

Shared Memory

- To destroy a segment:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- Shared memory segments stay persistent even after all processes have died!
  - You must destroy them in your programs
  - The ipcs command shows existing segments (and semaphores)
  - You can destroy them by hand with: *ipcrmshm* < *id* >

Shared Memory

Example

```
int main() {
  int shmid = shmget(IPC_PRIVATE, sizeof(int), 0600);
  int *shared_int = (int *) shmat(shmid, 0, 0);
  *shared_int = 42;
  if (fork()==0) {
    printf("The value is: %d\n", *shared_int);
    *shared_int = 12;
    shmdt((void *) shared_int);
  }
  else {
    sleep(1);
    printf("The value is: %d\n", *shared_int);
    shmdt((void *) shared_int);
    shmctl(shmid, IPC_RMID, 0);
  }
}
```

Threads vs. Processes

- Multi-process programs are expensive:
  - fork() needs to copy all the process' memory, etc.
  - Inter-process communication is hard
- Threads: "lightweight processes"
  - One process contains several "threads of execution"
  - All threads execute the same program (but can be at different stages within it)
  - All threads share process instructions, global memory, open files and signal handlers
  - But each thread has its own thread ID, stack, program counter and stack pointer, errno and signal mask
  - There are special synchronization primitives between threads of the same process

Threads in C and Java

- Threads in C:
  - Posix threads (pthreads) are standard among Unix systems
  - The operating system must have special support for threads
  - Programs must be linked with -lpthread
- Threads in Java:
  - Threads are a native feature of Java: every virtual machine has thread support
  - They are portable on any Java platform
  - Java threads can be mapped to operating system threads (native threads) or emulated in user space (green threads)

Creating a Pthread

- To create a pthread:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
void *(*start_routine)(void *), void *arg);
```

  - thread: thread id
  - attr: attributes (i.e., options)
  - start routine: function that the thread will execute
  - arg: parameter to be passed to the thread

- To initialize a pthread attribute:

```
int pthread_attr_init(pthread_attr_t *attr);
```

Stopping a Pthread

- A pthread stops when:
  - Its process stops
  - Its parent thread stops
  - Its start routine function returns
  - It calls pthread exit:

```
#include <pthread.h>
void pthread_exit(void *retval);
```

- Like processes, stopped threads must be waited for:

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

Pthread Create/Delete Example

```
#include <pthread.h>
void *func(void *param) {
int *p = (int *) param;
printf("New thread: param=%d\n",*p);
return NULL;
}

int main() {
pthread_t id;
pthread_attr_t attr;
int x = 42;
void pthread_exit(void *retval);

pthread_attr_init(&attr);
pthread_create(&id, &attr, func, (void *) &x);
pthread_join(id, NULL);
}
```

Detached Threads

- A "detached" thread:
  - Does not need to be pthread join()ed
  - Does not stop when its parent thread stops
- By default, threads are "joinable" (i.e., "attached")
- To create a detached thread, set an attribute before creating the thread:

```
pthread_t id;
pthread_attr_t attr;

 pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
pthread_create(&id, &attr, func, NULL);
```

- You can also detach a thread later with pthread detach()

Pthread Synchronization with Mutex

- Pthreads have two synchronization concepts: mutex and condition variables
- Mutex: mutual exclusion

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutex_init(pthread_mutex_t *mutex,
const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Pthread Synchronization with Mutex

- Pthreads have two synchronization concepts: mutex and condition variables
- Mutex: mutual exclusion

```
pthread_mutex_t array_mutex;
int add_elem(int elem) {
int n;
pthread_mutex_lock(&array_mutex);
if (nbelems==32) { pthread_mutex_unlock(&array_mutex); return -1; }
array[nbelems++] = elem;
n = nbelems;
pthread_mutex_unlock(&array_mutex);
return(n);
}
int main() {
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutex_init(&array_mutex, &attr);
...
pthread_mutex_destroy(&array_mutex);
}
```

Creating Java Threads

- A Java thread is a class which inherits from Thread
- You must overload its run() method:

```
public class MyThread extends Thread
 {

   private int argument;
   MyThread(int arg) {
     argument = arg;
   }
   public void run() {
     System.out.println("New thread started! arg=" + argument);
   }
}
```

- To start the thread:

```
MyThread t = new MyThread(42);
t.start();
```

Stopping Java Threads

- A Java thread stops when its run() method returns
- You do not need to join() for a Java thread to finish

```
MyThread t = new MyThread(42);
t.start();
...
t.join();
```

Java Thread Synchronization with Monitors

- A monitor is similar to a mutex:

```
public class AnotherClass {
synchronized public void methodOne() { ... }
synchronized public void methodTwo() { ... }
public void methodThree()
{ ... }
}
```

- Each object contains one mutex, which is locked when entering a synchronized method and unlocked when leaving
  - Two synchronized methods from the same object cannot be executing simultaneously
  - Two different objects from the same class can be executing the same synchronized method simultaneously

Java Thread Synchronization with Monitors

- So, the previous class is equivalent to:

```
public class AnotherClass {
private Mutex mutex;
public void methodOne()
{ mutex.lock(); ...; mutex.unlock(); }
public void methodTwo()
{ mutex.lock(); ...; mutex.unlock(); }
public void methodThree() { ... }
}


...except that the Mutex class does not exist!
```

Condition Variables in Java

- There is no real condition variable in Java
- But you can explicitly block a thread
  - All Java classes inherit from class Object:

```
class Object {
void wait();
/* blocks the calling thread */
void notify();
/* unblocks one thread blocked in this object */
void notifyAll(); /* unblocks all threads blocked in the object */
...
};
```

- wait(): causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
- notify(): wakes up a single thread that is waiting on this object's monitor.
- notifyAll(): wakes up all threads that are waiting on this object's monitor.

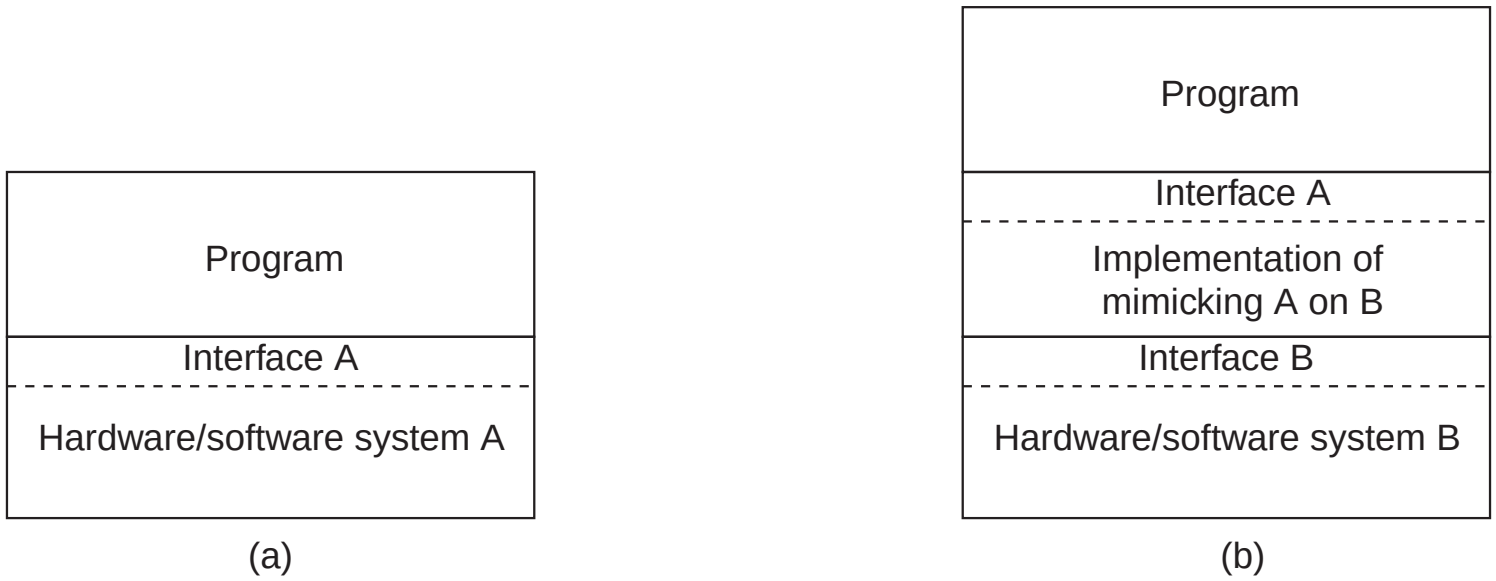
Condition Variables in Java

- This means that each object contains one (and no more) condition variable
- The wait(), notify() and notifyAll() methods must be called inside a monitor

Virtualization

Virtualization is becoming increasingly important:

- Hardware changes faster than software
- Ease of portability and code migration
- **Isolation** of failing or attacked components



Architecture of VMs

Virtualization

Virtualization can take place at very different levels, strongly depending on the **interfaces** as offered by various systems components: