Processes

- What is a process?
  - The unit of execution of the operating system
    - Unit of scheduling
    - Runtime execution context
    - Unit of resource accounting

- A process consists of
  - An address space
  - Code
  - Data
  - Stack pointer and program counter
  - Registers and content
  - Resources

Concurrent Programming

Incrementing a global counter

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>i++</td>
<td>i++</td>
</tr>
</tbody>
</table>

Process A

- fetch i: 0
- increment
- store back i

Process B

- fetch i: 0
- increment
- store back i

fork

- fork()
  - Creates a new process and returns its PID
  - Process receives a copy of the process state
  - Both processes (parent and child) will resume with the next command
  - Child will “see” pid being 0.

```c
int pid = 0;
pid = fork();
if (pid == 0)
    ChildProcess();
else
    ParentProcess();
```
wait

- `wait(status)`
  - Waits for a (any) child to terminate and sets status to the return value.

- `waitpid(pid, status, options)`
  - Returns `ECHILD` if there are no “unwaited-for” children left.

Example: Signals

```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

void sighup_handler()
{
    printf("the child has received a SIGHUP\n");
}

int main()
{
    int child_pid = 0;
    child_pid = fork();
    printf("the process id is %d and the child process id is %d\n", getpid(), child_pid);
    if (child_pid == 0)
    {
        // I am a child
        signal(SIGHUP, sighup_handler);
        printf("the child has registered its handler\n");
        for (;;) {} 
    } else
    {
        // I am the parent
        kill(child_pid, SIGHUP);
        printf("the parent has issued the signal\n");
    }
    return 0;
}
```

IPC

- Shared Memory
  - Set up under a shared key
  - Gets mapped into address spaces

- Named Pipes (FIFO)
  - File descriptor

- Sockets
  - `localhost`

Threads

- Lightweight unit of execution
  - Threads share the address space

  - **Unix: pthread**
    - `pthread_create`
    - `pthread_join`

  - **Java: Thread**
    - class `Thread`
      - `public void run()`
      - `public void start()`
Threads in Scala

```scala
val t = new Thread(new Runnable {
    def run() {
        println("this is running concurrently");
    }
})
t.start
```

Coordination

- In the presence of concurrent access, we need to coordinate
- Locks, Mutexes
- Condition Variables
- Semaphores
- Monitors

Dining Philosophers

- Philosophers think
- Thinking makes them hungry
- When they want to eat, they need to grab the fork left and right of them
- Problem: Starvation
  - Liveness vs. Safety

Locks and Mutexes

- Spinlock
  - while (test_and_set(lock)) {
  
- Mutex
  - pthread_mutex_t mutex;
  - pthread_mutex_init(&mutex, NULL);
  - pthread_mutex_lock (&mutex);
  - pthread_mutex_unlock (&mutex);
Locks in Java/Scala

- **Example: Java ReadWriteLock**

```java
private final ReadWriteLock lock = new ReentrantReadWriteLock();
lock.writeLock().lock();
try {
    if (callbacks == null) {
        callbacks = new ArrayList<CallbackRegistration<V, ?>>();
    }
    callbacks.add(new CallbackRegistration<V, T>
        (callback, tag, oneTime, freshnessRequirement));
} finally {
    lock.writeLock().unlock();
}
```

Semaphores

- Locks provide mutual exclusion but no execution order
- **Example: Producer/Consumer with bounded queue**

```java
Semaphore is a synchronization primitive that can be incremented (post) and decremented (wait).
Decrement sleeps until the semaphore becomes positive

sem_t sem;
sem_init(&sem, 0, 5);
sem_post(&sem);
sem_wait(&sem);
```

Condition Variables

- Wait for some event to happen
- Simple inter-thread communication schema
- Integrated with mutex

```c
pthread_mutex_t mutex;
pthread_cond_t cv;

pthread_mutex_lock(&mutex);
pthread_cond_wait(&cv, &mutex);
... do stuff...
pthread_mutex_unlock(&mutex);
```

Condition Variables

- Other thread:

```c
pthread_mutex_lock(&mutex);
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mutex);
```
Critical Sections

- Java: `synchronized`

```java
synchronized(list) {
    list.add("item");
    if (list.size() > 10) return;
}
MONITORENTER
MONITOREXIT
```

Monitors

- Protection around objects
- No more than one thread is active within a monitor
- Threads within the same monitor can signal (like for condition variables)

Monitors in Java/Scala

```java
Object o;

synchronized(o) {
    o.wait();
}

Other thread:
    synchronized(o) {
        o.notifyAll();
    }
```

Amdahl's Law

\[ S(n) = \frac{1}{(1-P) + \frac{P}{n}} \]

Speedup as a function of the degree of parallelism

Lock-Free Data Structures

- Locking entire sections of the code is problematic
  - Hinders parallelism
  - Priority inversion
  - Preemption

- Designing lock-free algorithms is very hard
  - Idea: design lock-free data structures

Example: Lock-Free Linked List

- CAS for changing pointers
- Delete: logical delete, then physical delete

OpenCL

- Portable framework for programming heterogeneous devices
- Mostly used for GPGPUs
- Library and external DSL based on C99
- Data-parallel computing in a task-based fashion

Example: Cell
OpenCL
- One host, multiple compute devices
- Devices have local memory, host has local memory

SIMD
- Single Instruction, Multiple Data
- Example:

Programming in OpenCL
- Idea: replace loop with kernel function
- Traditional CPU loop
  ```
  for (int i=0; i<DIM, i++) {
      c[i] = a[i] + b[i];
  }
  ```
- Kernel function
  ```
  vadd(...) {
      int id = get_global_id(0);
      c[id] = a[id] + b[id];
  }
  ```
Programming in OpenCL

Kernel
- Unit to handle data parallelism

```c
__kernel void vmul(__global const float *a, __global const float *b, __global float *c) {
    int gid = get_global_id(0);
    c[gid] = a[gid] * b[gid];
}
```

- No function pointers
- No variable-length arrays and structures
- No recursion

OpenCL Memory Model

Kernel
- Data types
  - Scalar
    - char, uchar, short, ushort, int, uint, long, ulong, float
  - bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void
  - Image
    - image2d_t, image3d_t, sampler_t
  - Vectors
    - Length 2, 3, 4, 8, 16
    - Endian-safe, aligned at vector length

- Synchronization primitives
  - mem_fence
  - barrier
Host Application

- Get the platform
  - `err = clGetPlatformIDs(1, &platform, NULL);`
- Get the device
  - `err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &dev, NULL);`
- Create a context
  - `context = clCreateContext(NULL, 1, &dev, NULL, NULL, &err);`
- Create command queue
  - `queue = clCreateCommandQueue(context, dev, 0, NULL);`

Host Application

- Create program from source
  - `program = clCreateProgramWithSource(context, 1, (const char**) &src_str, &src_len, &err);`
- Build program
  - `err = clBuildProgram(program, 0, NULL, NULL, NULL);`
- Create the kernel
  - `kernel = clCreateKernel(program, "vmul", &err);`
- Set kernel arguments
  - `err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_in);`

Host Application

- Enqueue data to be written
  - `err = clEnqueueWriteBuffer(queue, a_in, CL_FALSE, 0, sizeof(float) * LENGTH, a_data, 0, NULL, NULL );`
- Enqueue kernel
  - `err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, (const size_t*) &count, NULL, 0, NULL, NULL);`
- Enqueue read
  - `err = clEnqueueReadBuffer(queue, c_out, CL_TRUE, 0, sizeof(float) * LENGTH, c_res, 0, NULL, NULL);`