Threads 1

pthreads

Concurrency
processes & threads

Process: fundamental unit of execution
- fork() - create new process
- exec() - xfer ctl to an exec image
- wait() - for child to exit

OS context switch chooses active proc depending on #cores

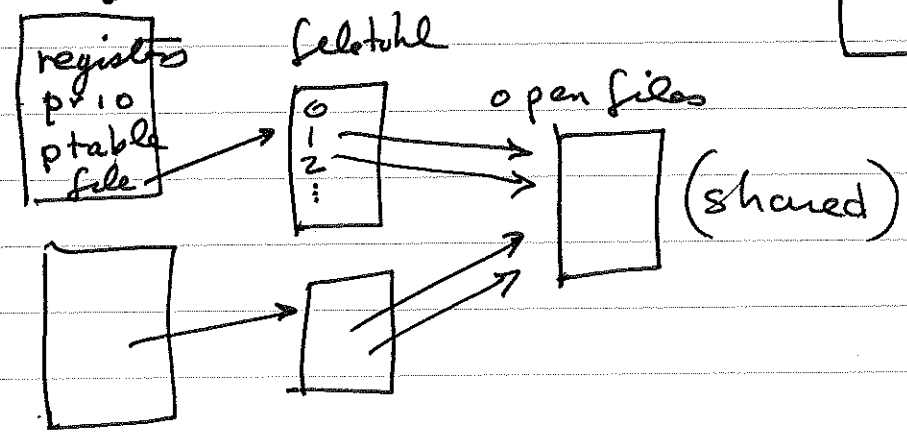
concurrency - apparent parallelism
parallelism - proc/threads run on cores (CPUs)

process termination
- exit (0) or exit (1...255)
  - = bash uses exit(128) for failed exec
  - = bash uses exit(128+signo) for crash
- killed by a signal
  - internal: segfault, segzdiv
  - external: sigkill, sigint, sigterm

context switch (proc or thread)
- save registers
- load reg.
- xfer control
- each thread has own stack
- each proc has own resources
  - memory
  - open file desc — file table
  - system locks

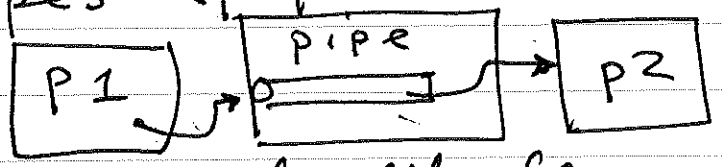
Threads 2

process



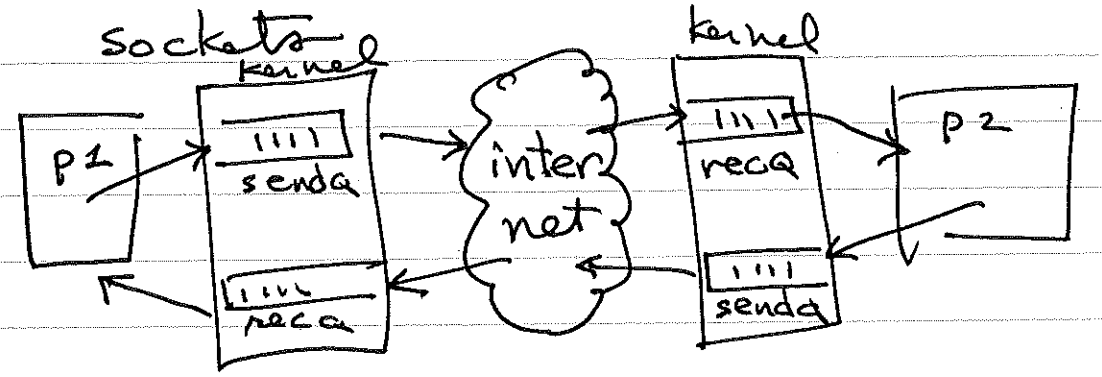
communication ← kernel.

pipes (popen)



looks like file IO

sockets



Threads

- "lightweight" processes
- create: much faster

pthreads — green — preemptive
Kernel — non preemptive

Threads

Threads 3

- share process
  - address space
  - open files
  - OS file locks
  - etc.

green threads: managed by libraries
kernel threads: OS heavy threads

blocking syscalls – entire proc is blocked awaiting response

each thread has own:
- registers
- stack

shares:
- addr space
- global (static) vars
- open files
- child procs
- signals

Threads 4

Race conditions
- shared memory

ex: observer / reporter

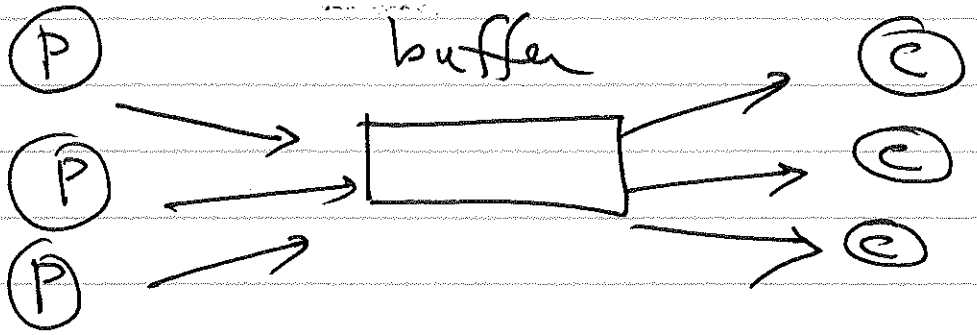
```
obs: loop{
       wait event
       [[c = c+1 ]]
     }

rep: loop {
       wait n time
       [[x = c;
         c = 0 ]]
       print x
```

Critical section
- bound shared var
- no 2 threads simultaneous in crit sec
- no assumption about speed of CPU or sched
- no p/thread of crit may block
- no thread wait forever (starvation)

mutex (mutual exclusion)
- disable intr (kernel only)
- lock variables
- strict alternation (client-serv ex)
- busy wait (wastes CPU)
- TSL
- sleep/wake

Threads 5

producer/consumer



AKA bounded-buffer (Queue)

```
prod:                          cons {
loop {                         loop {
  make x                         if (empty) wait
  if (full) wait                 consume x
  put x                          wake (P)
  wake (c)                       use x
}                              }
```

Semaphores

E.W. Dijkstra

P = proberen = down (try)
V = verhogen = up (raise)

```
P(s): if (s==0) wait
      s = s - 1

V(s): s = s + 1
      if (s == 0) wake one
```

Thread 6

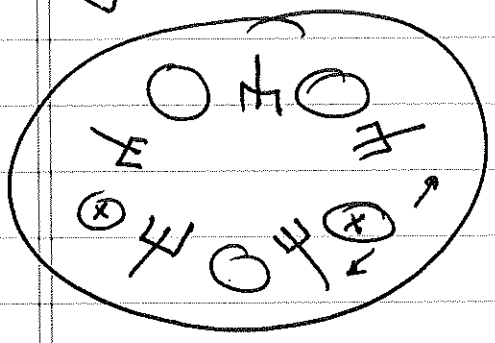
Mutex

ensure mutual exclusion
lock() - only on thread gets in
unlock() - gets out; allows others

Conditional Variables

wait() - for some conditional
- suspended
signal() - allow wait to continue

Dining Philosophers



```
philo {
  loop {
    think()
    pick leftfork
    pick rightfork
    eat()
    drop leftfork
    drop rightfork
  }
```

- deadlock
- Starvation

Phils

```
loop { think()
       take(i)
       eat()
       drop(i)
     }
take (i) {
    down(&mtx)
    state[i] = hungry
     test(i)
    up( &mtx)
    down(& s[i])
}
test (i)
    if( s[i] = H and s[L] = E & s[R] = E)
                s[i] = E
              up (s[i])
    }
```

pass salt for prio

prod/cons sema?

Thread 7

```
state [N]
mutex = 1
sema ph s [N]
drop(i) {
    down (&mtx)
    s[i] = think
    test (L)
    test(R)
    up (&m)
```

Readers / Writers

Thread 8

many readers in DB
one writer in DB

```
writer(){
    make data
    down (db)
    write
    up (db
}
```

```
reader {
    down mutex
    R = R + 1
    if (R == 1) down db
    up mutex
    read ...
    down mutex
    R = R - 1
    if (R == 0) up DB
    up (mutex)
    use data
}
```

Deadlock

Thread 9

each proc/thread waits for another

ex A: lock l1
lock l2

B: lock l2
lock l1

resource deadlock

other problems

livelock

```
A: while( ){
     lock l1 if available
     if not continue
     :
   }
```

race cond - unsynch.

starvation: never gets resource

(Java uses "synchronized")

Semaphores

Thread 10

value > 0 = # let through gate
= 0 = gate is locked
< 0 = # threads waiting
semaphore lock(1) is ≡ mutex

```
class semaphore {
  private:
    mutex lock;
    condition_variable cond;
    int value {};
  public:
    semaphore (int val): value(val) { }
    void down() { // "proberen"
        unique_lock<mutex> ulock{lock};
        --count;
        if(count < 0) cond.wait();
    }
    void up() { // "verhogen"
        unique_lock<mutex> ulock{lock};
        if(count < 0) cond.notify_one()
        ++count
    }
}
```

Producer/Consumer

```
semaphore lock{1};
semaphore empty{N};
semaphore full{0};
```

```
Producer:
loop{
  produce(X)
  empty.down()
  lock.down()
  Q.insert(x)
  lock.up()
  full.up
}
```

```
consumer
loop{
  full.down
  lock.down
  x=Q.remove()
  lock.up()
  empty.up()
  consume(
}
```