Interprocesses Communication:

- Shared Memory

1. systemcall to create shared memory with fixed size
2. P1 & P2 can r/w to shared memory in user mode
3. Requires synchronisation protocol so that processes do not overwrite each other (eg P1 write faster than P2 reading, eventually overwrote)

- Socket (array location in kernel space) SLOWER

1. requires sys call to read & write
2. Kernel will automatically synchronise process communication

* system call is exp, but sync is troublesome (things can go wrong)

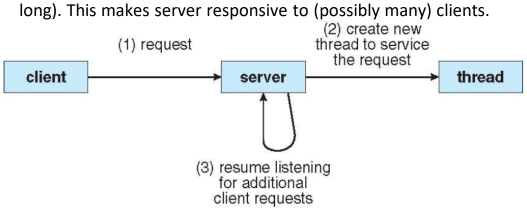
Process couples (gives) **concurrency** (interleaved execution) & **protection** (address space isolated from one another, each process in its own VM) {expensive}

Thread gives concurrency without protection

Each thread has own reg&stack within process' space. Shares code, data & files. Eg. if thread 1 is modifying data, thread 2 will see changes (requires sync)

Multithreaded Architecture (one of the uses of threads)

Server's main thread creates new thread to handle request by client. Main thread can quickly go back to accepting new requests (instead of waiting for previous request to finish, which may take long). This makes server responsive to (possibly many) clients.



Thread Scheduler (from programming language library, done in user space): decides when threads are paused and resumed

Kernel doesn't know how many threads are there in the process (CPU user space), all it knows is the *number of processes*.

Type of Threads:

1. User Threads

Not known to kernel, scheduled by thread scheduler in thread library, runs in user mode, cheaper to create

2. Kernel threads (runs in kernel mode)

Has kernel data structure (way larger than user thread data struc): thread control block (exp to create & perform kernel thread's own context switch), scheduled and known by kernel

Note: If one kernel thread is blocked, the other kernel threads that are runnable can run. User mode cannot interrupt kernel mode. However within kernel mode itself, the kernel scheduler free to choose which kernel thread to run/pause/kill.

Note: Kernel core with >1 threads -> for task switching

Only the kernel threads will be able to update the process table

Kernel is **one single process** that have many threads (#kernel threads is OS specific & hardware supported)

Each thread in user that wants to make a sys call have to go through the kernel thread

(eg. 4 threads in P1 cannot make use of multicore, can only use 1 kernel thread) If want to use 4 cores, min #kernel threads = 4

1to1: most expensive

manytomany: don't wait just go do other process (flexible), make use of multicores (BEST model)

User Space

- Each process has at least one thread
- User threads are managed by thread library (C, Java, C#) in user space
- User P1 doesnt know P2 (between processes)
- Thread library is also responsible for thread mapping
- 2 processes run in parallel if kernel thread assigned are in different cores
- if T1 have a blocking syscall while T2 is ready in P1, P1 will be waiting when it is many-to-one configuration:

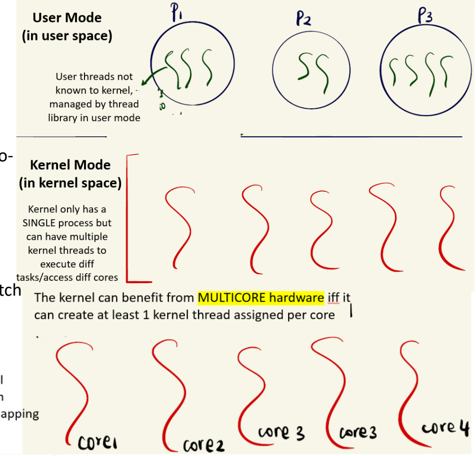
>Cannot benefit from multi-core (threads in same process)

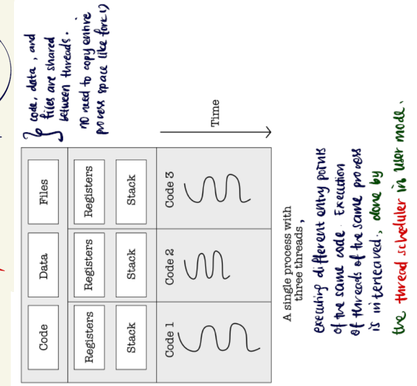
> If thread in a process makes blocking sys call, the whole Process is blocked.

Kernel (a single process)

- Proc Table doesn't contain information on threads
- can share 2 kernel threads with the same core (kernel thread scheduler will thread switch between them)

* With Threads, you can achieve concurrency (interleaved execution) -> might achieve parallelism if there is more than 1 CPU

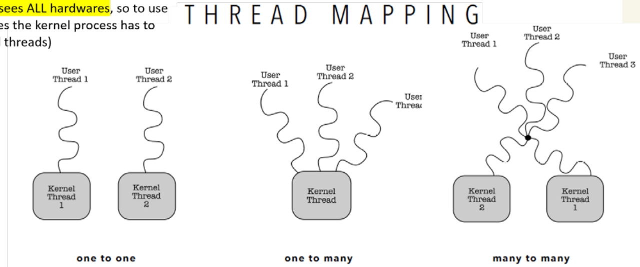




Note: Multiple kernel threads are required to achieve parallelism if the system has multiple cores. (because a single kernel process oversees ALL hardwares, so to use different cores the kernel process has to create kernel threads)

Thread mapping must be SUPPORTED by the programming language's API (thread library). If you want your app to make use of your multicores then choose a library that allows either **onetoone** mapping or **manytomany** mapping

THREAD MAPPING



one to one

✓ maps each user thread to kernel thread
✓ provides more concurrency than manytoone model
✓ allows multiple threads to run on multiprocessor
✗: kernel thread creation overhead. Developer may not be able to create too many user threads (Kernel threads are large in size)

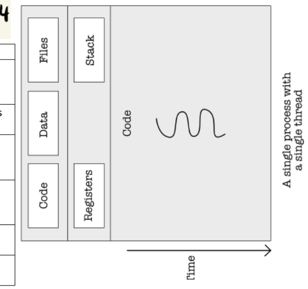
one to many

✓ No need for kernel thread scheduler
✓ CHEAP, but only 1 thread can access kernel at a time, multiple threads aren't able to run on multiprocessors
✗: An entire process will block if a thread in that process makes a blocking system call

many to many

For this model, a fixed kernel thread per machine or per app
✓ User can create as many user threads, but true parallelism (true concurrency) isn't gained because the kernel can only schedule a process -> then a user thread at a time

Processes	Threads
More overhead to create (bcus copying over entire VM space, including code, data & files)	Lesser overhead to create (shares data, code & files)
Requires private address space, has protection against other processes	Easier to create since they share address space, only differ in program execution
Processes are indpt of one another, no concurrency issues	Requires careful programming since threads of same process share data structures & hence are interdependent
No synchronisation overhead, easier to program and work as intended	Can potentially suffer from synchronisation overheads, harder to program & work as intended
Can benefit from parallel execution on multiprocessor system	same
A process can have many threads executing different tasks	Good for responsiveness of a process



- *Kernel* threads
  - Known to OS kernel
  - Scheduled by kernel CPU
  - Take up kernel data structure (e.g., Thread Control Block like PCB)
  - More expensive
- *User* threads
  - Not known to OS kernel
  - Scheduled by thread scheduler (running in user mode) in thread library (e.g., POSIX pthread or Java threads) linked with process
  - Less expensive

Thread vs. Process

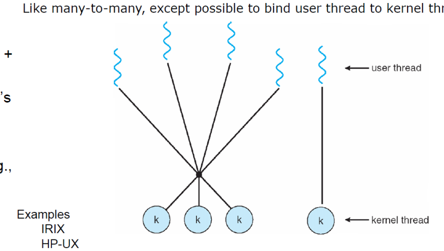
- Recall: Process couples two abstractions: concurrency and protection
- Can I decouple the two, e.g., have concurrency *without* protection?
  - Yes, use *threads* (thread = line of execution, has registers + stack)
  - Many threads can run within a process, share the process's address space
    - No protection between them
    - But "IPC" (or inter-thread communication) is simple (e.g., no need for shmget, shmat, etc) and fast (much less to save/restore at context switch)

Why (or why not) threads?

- Speedup by parallel execution (e.g., your Lab 2)
  - On multiprocessor or multicore systems
- Responsiveness
  - While one thread is blocked for IO, another thread can be executing and doing useful computation
- Logical modularity
  - Though without fault isolation
- Disadvantages
  - Context switch + synchronization overheads
  - Can be much harder to program and get right!

Hybrid Model

Like many-to-many, except possible to bind user thread to kernel thread



Examples
IRIX
HP-UX

The integer variable times is shared between processes P and Q. The C (or Java) instruction times++ is implemented as the following sequence of three atomic machine instructions (R1=times; R1=R1+1; times=R1;) Initially, times is 10. Then each of P & Q executes the times++ instruction. (a) show an interleaving execution order of 6 machine inst st times will have value of 11 finally. **1. P: R1 = times; 2. P: R1 = R1 + 1; 3. Q: R1 = times; 4. P: times = R1; 5. Q: R1 = R1 + 1; 6. Q: times = R1;** (alternative answers, e.g., P interchanged with Q, Steps 4 and 5 interchanged,)
(b) What are all the other possible final value(s) of times? **12**

2. A process P needs to transfer 1 Kbytes of data stored in a local buffer to another process Q.
(a) If message passing is used, what is the minimum needed (i) number of system calls, and (ii) amount of data copied? **(i) two; (ii) 2Kbytes.**
(b) Repeat Question (a) if shared memory is used. You can ignore the overhead of setting up the shared memory between P and Q. **(i) zero; (ii) zero.**
3. A multithreaded process P contains two user-level threads T1 and T2. Is each of the following statements true or false?
(a) It is possible for a bug in T1 to corrupt the stack of T2. **True**
(b) Context switching between T1 and T2 is performed by code running within P. **True**

1. Threads are easier to create than processes since they don't require a separate address space.

2. Multithreading requires careful programming since threads share data strucures that should only be modified by one thread at a time. Unlike threads, processes don't share the same address space.
3. Threads are considered lightweight because they use far less resources than processes.
4. Processes are independent of each other. Threads, since they share the same address space are interdependent, so caution must be taken so that different threads don't step on each other. This is really another way of stating #2 above.
5. A process can consist of multiple threads.

Producer-Consumer Problem

- A producer puts a new item of work into a shared buffer
- A consumer takes an item of work from the buffer
- The buffer can store a fixed (finite) number of items, i.e., *bounded buffer*
- If producer and consumer run as separate processes (or threads), how can we *ensure* that their concurrent execution is correct?
  - Each process/thread makes *non-zero progress*
  - But otherwise, can't assume anything about their relative speed of execution

Race Condition

- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2
- Assume 1 CPU. Execution may interleave as (initially, count = 5):
  T0: producer execute register1 = count {register1 = 5}
  T1: producer execute register1 = register1 + 1 {register1 = 6}
  T2: consumer execute register2 = count {register2 = 5}
  T3: consumer execute register2 = register2 - 1 {register2 = 4}
  T4: producer execute count = register1 {count = 6 }
  T5: consumer execute count = register2 {count = 4}

  Is execution correct? Why?

NB: each process has own registers (as usual); **count** is shared. The instructions load, store, and arithmetic operations are *atomic* (i.e., can't be interrupted in the middle).

The Critical Section Problem

- Need to (at least) guarantee *mutual exclusion* in updates to **count** variable for code on Slides 6.3 and 6.4
  - If one process is in the middle of updating count, no other processes can be updating count at the same time
- Update needs to be protected in a *critical section* of code
  - A critical section is a segment of code (e.g., sequence of instructions that complete **count++** on Slide 6.5)
  - Processes can't be inside their critical sections at the same time (only *one* process can be)

*NB: We use **process** in our discussions; same ideas apply to **threads** as well*

Solution to Critical-Section Problem

A really correct solution should satisfy *all* these properties ...

1. **Mutual Exclusion** - If process P_i is executing in its critical section (CS), then no other processes can be executing in their critical sections.
   1. *Safety* property: something bad (more than one processes in CS) can't happen
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
   2. *Liveness* property: something good (a process entering CS) will eventually happen. *Mutual exclusion is trivial to satisfy without progress – why?*
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a **nonzero speed**
   - No assumption concerning relative speed of the processes (e.g., process P can run at same, much higher, or much lower speed than process Q)

NB: each critical section is of *finite length* (process will exit it after finite number of instructions – e.g., can't loop forever)

Synchronization Hardware

- Many systems provide *hardware* support for critical section code
  - Solutions become easier with hardware support
- Mutual exclusion on uniprocessors – could disable interrupt on process j's entry (into CS) and restore interrupt on j's exit (from CS)
  - j must execute CS in *entirety* before any other process can have a chance to run – why?
  - Doesn't work in general for multiprocessors – why?
- Many modern machines provide special *atomic hardware instructions*
  - Two common instructions
    - Test original value of memory word and set its value
    - Swap two memory words

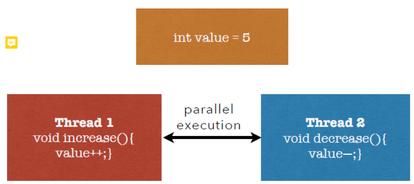
Definition of Hardware Instructions

```
public class HardwareData
{
   private boolean value = false;

   public HardwareData(boolean value) {
     this.value = value;
   }

   public boolean get() {
     return value;
   }

   public void set(boolean newValue) {
     value = newValue;
   }

   public boolean getAndSet(boolean newValue) {
     boolean oldValue = this.get();
     this.set(newValue);

     return oldValue;
   }

  public void swap(HardwareData other) {
    boolean temp = this.get();

    this.set(other.get());
    other.set(temp);
  }
}
```

*NB: **getAndSet** is guaranteed to be atomic by hardware, although it consists of multiple instructions; same for **swap***

THE RACE CONDITION

Behaviour of a system where its output is dependent on the sequence or timing of **uncontrollable** events



Possible outputs: value is 4, value is 5, and value is 6, depending on order of execution by scheduler (hence it is uncontrollable)

Count ++ and count - - are **not** atomic in machine language. If they are somehow atomic by hardware implementation (atomic : implemented in **one** clock cycle), then race condition would not have surfaced.

CRITICAL SECTION

Count ++	Count --
LD(count, Rx)	LD(count, Rx)
ADDC(Rx, 1)	SUBC(Rx, 1)
ST(Rx, count)	ST(Rx, count)

These have to be uninterruptible

Rules of critical sections:

- **Mutual exclusion:**
  - preventing race condition
- **Has progress:**
  - if no thread / process in CS, then
  - select process in queue that can enter CS as soon as possible
- **Bounded waiting:**
  - each CS has finite length
  - there's max number of times other thread / processes are allowed to "cut" queue