Sequntial Process
vs.
Co-operating Processes

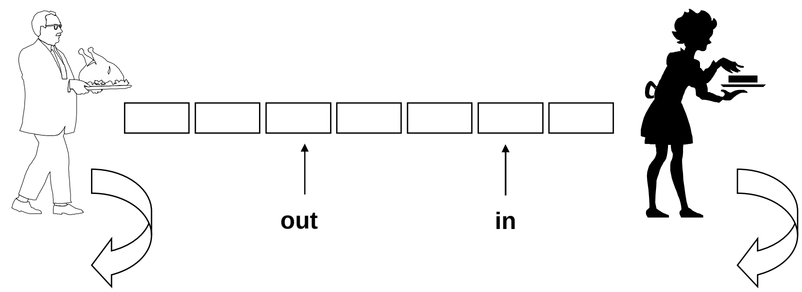
Example of a Sequential Process

```
/*Sequential Producer & Consumer*/

int i=0;

repeat forever

    Gather material for item i;
    Produce item i;
    Use item i;
    Discard item i;
    I=I+1;

end repeat
```

- **Analogy:**
  - ♦ Manufacturing and distribution
  - ♦ Print shops
  - ♦ Bank transaction
  - ♦ Airline reservation
  - ♦ Compiler Assembler
- **Problems:**
  - ♦ A simple process always running in sequence can be very inefficient.
  - ♦ All situations can not be modeled as a sequential process

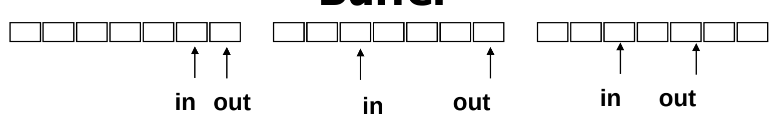
Example of Cooperation: Producer -Consumer problem

- **Producer produces information**
- **Consumer consumes information**



Operating Systems Lecture 7

Process Communication by Bounder Buffer




```
Var buffer array[0..n-1]
in=0;
out=0;

/*producer*/

repeat forever
..
Produce item nextp;
..
While( n+1 mode n == out)
    do nothing
    buffer[in]=nextp;
    in = in+1 mod n
end repeat
```

```
/*consumer*/

repeat forever

While( in == out)
    do nothing
    nextc=buffer[out];
    out = out+1 mod n
    ..
    Consume item nextp;
    ..
end repeat
```

Cooperating Processes

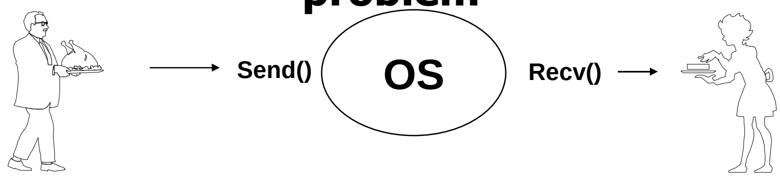
- Processes can run independently, But...
- In many situations it is advantageous if processes can work together:
  - ♦ Information sharing
    - » many user may want to access same info at the same time
    - » multiple write problem.
  - ♦ computational speedup
    - » A single job can be divided into concurrent tasks and each task can run in parallel while communicating occasionally.
    - » producer-consumer problem (uncompress-print, compiler-assembler cases).
  - ♦ Modularity

Inter process Communication (IPC)

- This is a facility that OS provides so that co-operating processes can communicate with each other more easily.
- Goal: Save processes from buffer management, synchronization.

- Blocking Send:
  - ♦ send(destination_process, message)
  - ♦ Sends a message to another process then blocks until message is received.
- Blocking Receive:
  - ♦ receive(source_process, message)
  - ♦ Blocks until the message is received.

IPC Solution to Producer-Consumer problem




```
/*producer*/

repeat forever

Produce item nextp;
Send(consumer, nextp)
end repeat
```

```
/*consumer*/

repeat forever

receive(producer, nextc)
consume item nextp;
end repeat
```


Operating System Headaches

- **How links are established?**
- **Can a link be associated with more than 2 processes?**
- **How many links can there be between each pair of processes?**
- **What is the capacity of a link? Any buffer? If so, how much?**
- **Can the message size vary?**
- **Is the link unidirectional or bidirectional?**
- **What to do if messages are lost?**
- **What to do if either sender or receiver dies?**

Few Design Choices

Direct vs. Indirect Communication

- **Direct Communication:**
  - explicitly name the other process
  - one link between two process
  - can be bidirectional
- **Indirect Communication:**
  - use mailbox owned by receiver
  - many can be send to one.
  - Receiver may change house

Variations in Buffering

- **Zero Capacity**
  - no message wait
  - sender or receiver one must wait
- **Bounded Capacity**
  - sender or receiver one must wait, if buffer is full
- **Unbounded Capacity**
  - sender can always continue

Direct Communication

Processes must name each other explicitly:

- send **(*P, message*) – send a message to process P**
- receive**(*Q, message*) – receive a message from process Q**

Properties of communication link

- **Links are established automatically.**
- **A link is associated with exactly one pair of communicating processes.**
- **Between each pair there exists exactly one link.**
- **The link may be unidirectional, but is usually bi-directional.**

Indirect Communication

Messages are directed and received from mailboxes (also referred to as ports).

- **Each mailbox has a unique id.**
- **Processes can communicate only if they share a mailbox.**

Properties of communication link

- **Link established only if processes share a common mailbox**
- **A link may be associated with many processes.**
- **Each pair of processes may share several communication links.**
- **Link may be unidirectional or bi-directional.**

Indirect Communication

Operations

- **create a new mailbox**
- **send and receive messages through mailbox**
- **destroy a mailbox**

Primitives are defined as:

send**(*A, message*) – send a message to mailbox A**

receive**(*A, message*) – receive a message from mailbox A**

Indirect Communication

Mailbox sharing

- ***P1, P2,* and *P3* share mailbox A.**
- ***P1*, sends; *P2* and *P3* receive.**
- **Who gets the message?**

Solutions

- **Allow a link to be associated with at most two processes.**
- **Allow only one process at a time to execute a receive operation.**
- **Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.**

Synchronization

Message passing may be either blocking or non-blocking.

Blocking **is considered** synchronous

Non-blocking **is considered** asynchronous

send **and** receive **primitives may be either blocking or non-blocking.**

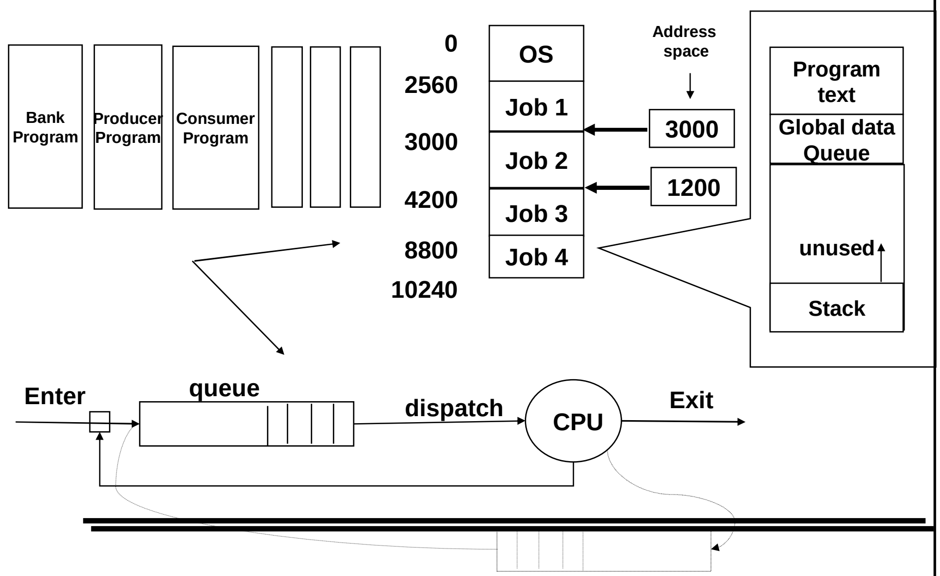
Threads

- **A simple process always running in sequence can be very inefficient. There is a need of dividing some jobs into multiple cooperating processes.**

- **Two many processes running concurrently, result in too much context switching.**

Solution: Thread

Unit of Scheduling & Unit of Resource Ownership



Conventional View of Process

A process is:

- ♦ **A unit of <u>resource ownership</u>**
  - » **A process has an address space**
  - » **A process has open files**
  - » **A process may hold IO devices**

It is also:

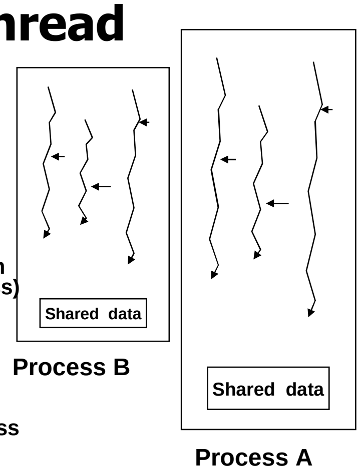
- ♦ **A unit of <u>scheduling</u>**
  - » **A process is the item of concurrent execution in the OS**
  - » **A CPU scheduler (dispather) assigns one process at a time onto a CPU.**

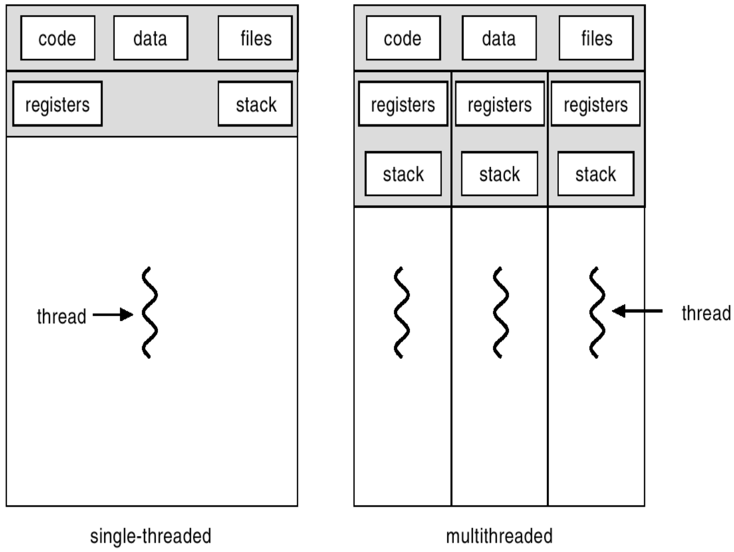
These two functions are usually linked together, but they don't have to. In modern OS:

- ♦ **Process=unit of resource ownership**
- ♦ **Thread=unit of scheduling**

Process vs. Thread

- **<u>Process = unit of resource ownership</u>**
  - ♦ **A process has:**
    - » **an address space**
    - » **program code**
    - » **OS resources (files, IO devices)**
- **<u>Thread=unit of scheduling</u>**
  - ♦ **A thread is a single sequential execution stream within a process (also called lightweight process)**
  - ♦ **A thread has:**
    - » **program counter**
    - » **stack pointer (SP)**
    - » **registers**
  - ♦ **A thread shares with other threads in the process group**
    - » **an address space**
    - » **program code, global variables**
    - » **OS resources (files, IO devices)**

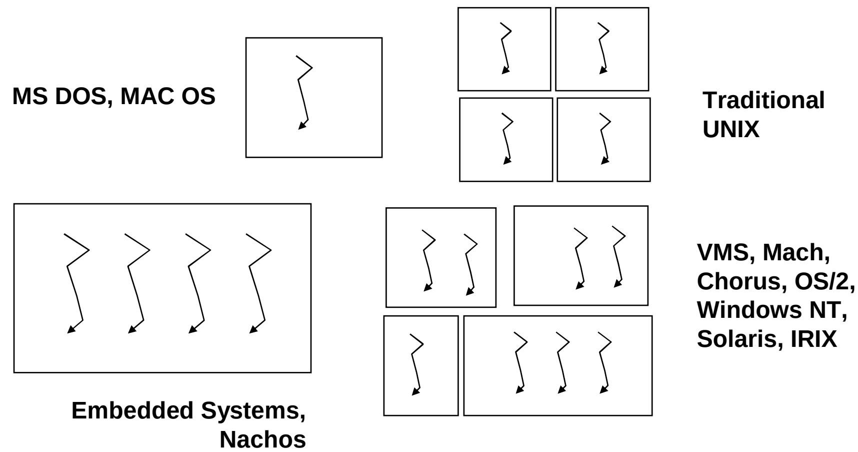




Process vs. Thread (contd..)

- **A thread is bound to a particular process**
  - ♦ **A process may contain multiple threads of control inside.**
- **All of the threads in a process:**
  - ♦ **can execute concurrently**
  - ♦ **share a common address space (and thus data other than private stack).**
- **Threads can block, create children, etc.**

Various Threaded Systems



Os-slide#21

Why Thread?

- **A process with multiple thread makes great server (printer server, file server, database server):**
  - ♦ **One server process, many 'worker' threads.**
  - ♦ **If one thread blocks (such as read request0, others can still continue executing.**
  - ♦ **All threads can share common data, no need for complicated inter process communication**
  - ♦ **also saves memory !**
- **But .. No protection between threads**
  - ♦ **since all threads in a process share common address space they can interfare with one another.**
  - ♦ **Generally all threads belongs to a single process so protection is not a big problem.**

Os-slide#22

User Level Thread

- User-level threads: a library of functions (to create, fork, switch, etc.) which user processes can call to create and manage their own threads.
- Positive Points:
  - Does not require OS modification
  - Simple representation- a PC, registers, stack and a small control block, all stored in the processes address space.
  - Fast- generally just a function call. No kernel intervention is needed.
- Negative Points:
  - OS has no knowledge of the threads so may get unfair attention
  - Requires non-blocking system calls (otherwise entire process blocks if a single thread blocks).
  - If one thread causes page-fault the entire process blocks.

Example: POSIX Pthread, Mach C-threads, Solaris 2 UI-threads

Kernel-Level Threads

- Kernel-level threads: kernel provides the system calls to create and manage threads.
- Positive Points:
  - kernel has full knowledge of all the threads. Scheduler can allocate more time to processes with more threads.
  - Good for applications that frequently blocks.
- Negative Points:
  - Slow: each system call is about 100 times slower.
  - Significant overhead and kernel complexity
  - requires a full TCB (thread control block) for all threads.

Examples: Windows NT, Windows 2000, Solaris 2 BeOS, Tru64 Unix

Mixed Support

Many system supports both user and kernel level thread. The mapping can be varied.

Many-to-One

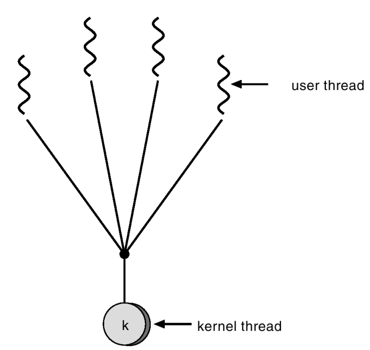
One-to-One

Many-to-Many

Many-to-One

Many user-level threads mapped to single kernel thread.

Used on systems that do not support kernel threads.

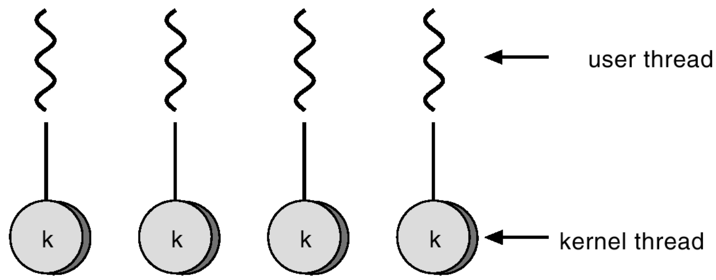


One-to-One

Each user-level thread maps to kernel thread.

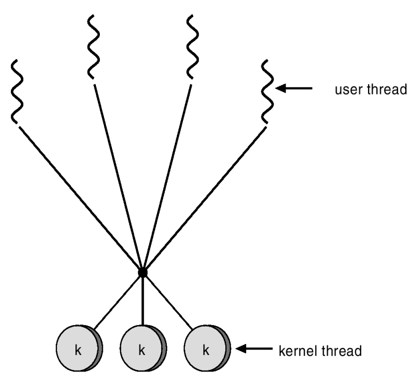
Examples

- **Windows 95/98/NT/2000**
- **OS/2**



Many-to-Many Model

- **Allows many user level threads to be mapped to many kernel threads.**
- **Allows the operating system to create a sufficient number of kernel threads.**
- **Solaris 2**
- **Windows NT/2000 with the *ThreadFiber* package**



Fork and Exec Calls in Thread

Fork

If one thread calls fork() does the new process duplicate all threads or the new process is single threaded?

Some Unix systems have both versions.

Execv

generally works in the same way. It will overwrite the entire process including all threads.

Thread Cancellation

How to handle cancellation?

Case A:

Multiple thread searching one DB. Then if one thread completes all can be terminated.

Case B:

multiple threads in an WebBrowser. One can be cancelled while others are running.

Thread Cancellation

Asynchrornous Cancellation

One thread immediately terminates the other.

Deferred cancellation

The target thread periodically checks if it should be terminated. It can orderly terminate.

“Cancellation point”

Solaris User Level Threads

User API

users create user level threads by “PThread” or “UI-thread” API for thread creation and management.

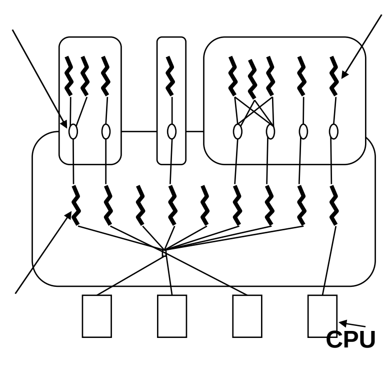
LWP

It has an intermediate level of thread called LWP– light weight process.

Each process contains atleast one LWP

The thread library mutiplexes user level threads on a pool of LWP.

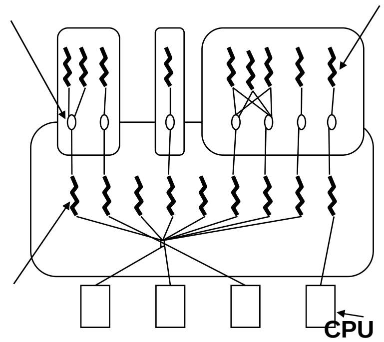
The user level thread who are currently on an LWP executes. Rests are blocked.



Solaris Kernel Threads

Standard Kernel Threads

- Executes all operations within kernel
- Each LWP has a kernel thread.
- Some kernel threads have no associated LWP and only does kernel job.
- Kernel level threads are only objects recognized by the scheduler.



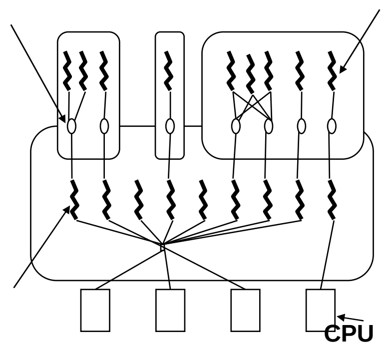
Thread Association

User level threads can be
  "bound" to a LWP. If bound only that thread will run on that LWP.

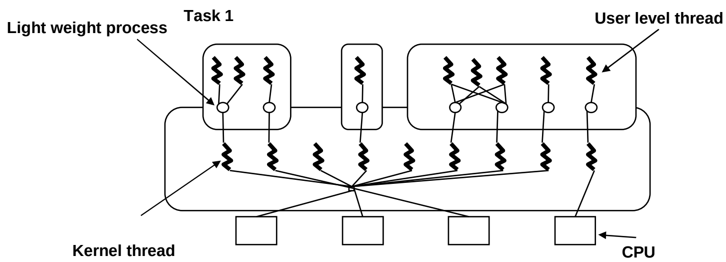
On request a LWP can be
  "dedicated" to one CPU.

Unbound threads
  of one application use a common pool of LWPs.

A group of threads in solaris (only)
  can all bind to one LWP.



Various Modes of Running



User level threads can be created easily.
Only when guaranteed concurrency with respect to a kernel event (such as an file I/O) will be needed one separate LWP will be needed.

Solaris threads Data Structures

User level thread
- **It has a thread ID, register set (PC, Stack Pointer), stack, priority (used by thread library).**
- **It is created by library call and Implemented in user space.**
- **Very fast.**

LWP
- **One register set for the user level thread now running. Memory and accounting information.**
- **A kernel data structure resides in kernel space.**

Kernel level thread
- **A small data structure and a stack. The data structure includes a copy of kernel registers, a pointer to LWP to which it is attached, and priority and scheduling information.**

Process
- **has everything described in PCB+ a pointer to a list of its threads.**