THREADS

* A process can be further divided into several units of execution called threads.
* Also called as light-weight process.
* Comprises:
  ↳ thread ID
  ↳ program counter
  ↳ register set
  ↳ stack
* threads of same process share:
  ↳ data section
  ↳ code section
  ↳ OS resources such as open files and signals.
* Thread is created only at run time - so it takes less time to be created than process creation.

BENEFITS OF MULTITHREADED PROGRAMMING:

i) Responsiveness
   ↳ if one thread is blocked, other threads are not affected.

ii) Resource sharing / communication
   ↳ Shared memory is default in thread and is fast.

iii) Economy
   ↳ It is more economical to create and context-switch among threads.

iv) Scalability
   ↳ Multithreading can be even greater in a multi-core architecture.

INTERLEAVING:

* Applied on single CPU core system.
* The CPU is shared by threads based on time or I/O
* Concurrency - illusion of simultaneously performing

OVERLAPPING:

* Applied on multi-core systems
* Execution of multiple threads is done simultaneously in multi cores.
* Parallelism

AMDHAL'S LAW:

* Identifies performance gain from adding additional computing cores.
* $$\text{Speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

S → serial code → that can be run one by one, not parallely.

N → No. of CPU's

CHALLENGES OF MULTITHREADING:

i) Identifying tasks
ii) Balancing
iii) Data Splitting
iv) Data dependency
v) Testing and Debugging

TYPES OF PARALLELISM:

i) Data Parallelism: Distributing subsets of the same data across multiple computing cores and performing the same operation on each core

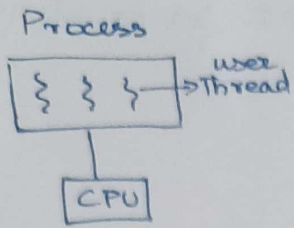
Eg: Sorting using Merge sort or Quick sort.

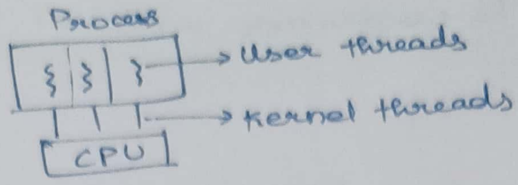
ii) Task Parallelism: Distributing tasks (threads) across multiple cores. Each thread performing a unique operation.

MULTITHREADING MODES:

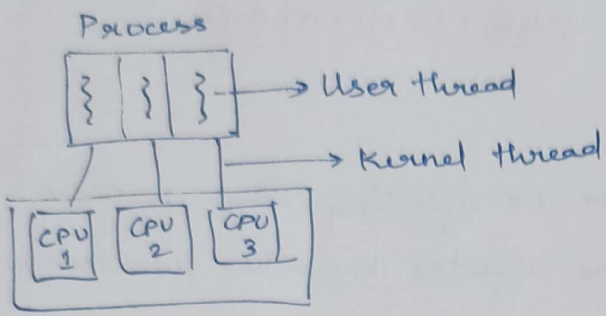
USER LEVEL THREADS	KERNEL LEVEL THREADS
i) Created & managed by the user	i) Created & managed by kernel
ii) Do not benefit from multicore	ii) Benefits from multicore system.
iii) Does not need mode switching	iii) Needs mode switching
iv) If a thread invokes IO operation, the kernel does not recognize threads, it treats like a process and other threads are also blocked	iv) Only that thread is blocked.
v) Implemented on any os	v) can be implemented only on OS that supports multi-threading.
vi) Eg: UNIX, Pthread, Javathread	vi) Eg: Windows, Linux, Mac OS x, Solaris

* One CPU :



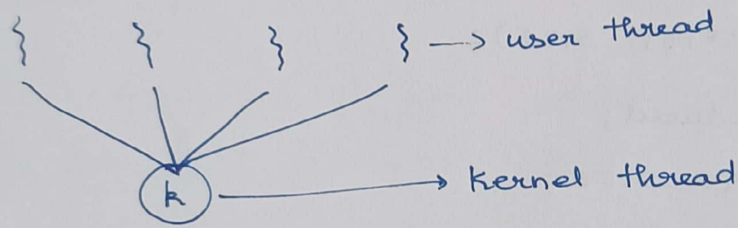


* Multicore :



MULTI THREADING MODELS:

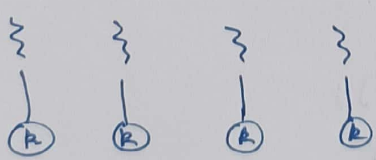
MANY TO ONE MODEL:



* OS considers as only one thread.

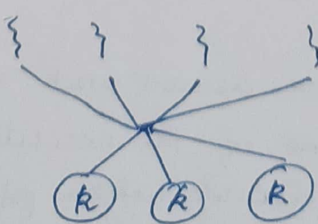
ONE TO ONE MODEL:

* More no. of threads burdens the application by creating more kernel threads



MANY TO MANY MODEL:

* Multiple ULT's will be confined to a certain number of KLT



THREAD LIBRARY:

* Thread library provides the programmer with an API for creating and managing threads.

i) POSIX Pthreads
ii) Windows - Kernel-level library
iii) Java. Pthreads

CONCURRENCY & SYNCHRONIZATION

TYPES OF PROCESSES:

i) Independent Process: without interaction or affecting other process.
ii) Cooperating Process: can affect or be affected by other processes.

PRODUCER CONSUMER PROBLEM:

PRODUCER:

```
while(true){
    while(counter == BUFFER_SIZE);
    buffer[in] = next_produced;
    in = (in + 1)% BUFFER_SIZE;
    counter++;
}
```

CONSUMER:

```
while(true){
    while (counter == 0);
    next_consumed = buffer[out];
    out = (out + 1)% BUFFER_SIZE;
    counter--;
}
```

RACE CONDITION:

* Situations where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.

SYNCHRONIZATION:

* Doesn't allow two processes to manipulate resources simultaneously
* Limits on number of simultaneous users

CRITICAL SECTION:

* Segment of code of the process which attempts to manipulate the shared resource.
* No two processes can be executing in their critical section at the same time simultaneously.

Process

remainder_section
critical section
entry_section
access shared resource
exit_section
remainder_section

* Requirements of a solution to critical-section problem:
  i) Mutual Exclusion - Only one process can perform critical section.
  ii) Progress - If no process is in its critical section, the intended one should be allowed.
  iii) Bounded waiting -

PETERSON'S SOLUTION:

* Software-based solution
* Does not work on modern architecture
* Restricted to two processes
* Data variables:
  ↳ int turn → equal to id of the process that enter the critical section.
  ↳ boolean flag[2] → indicate if a process wants to enter its critical section
  → if P_0 wants to enter, flag[0] is set true.

*

* Algorithm:

```
do{
    flag[i] = true ;
    turn = j ;
    while ( turn == j && flag[j]);
        critical section
    flag[i] = false ;
}while(true);
```

* satisfies all 3 conditions

SYNCHRONIZATION HARDWARE:

i) Compare & Swap Instruction

```
int compare_and_swap (int * word, int testval , int newval)
{
    int oldval ;
    oldval = * word;
    if (oldval == testval) ~~*word = newval;~~
        ~~return old~~ *word = newval ;
    return oldval ;
}
```

```
void p(){ while ((c_a_s( bolt, 0, 1) == 1) ; cs } bolt = 0 ; }
```

ii) Test & set() :

```
boolean test_and_set( boolean * target) {
    boolean rv = * target ;
    * target = true ;
    return rv ;
}
do { while ( test_and_set ( & lock));
        critical section
        lock = false ;
        remainder section .
}while(true);
```

* satisfies mutual exclusion
* Does not satisfy bounded waiting.

iii) Mutex Locks :

* process must acquire the lock before entering a critical section ; it releases the lock when it exits the critical section.

* Available = true

```
acquire ()
{
    while (! available) ;
        available = false ;
}
release()
{
    available = true ;
}
```

* Also called as spinlock

iv) Semaphores :

```
semaphore
+-----------------+
| private:        |
|     value;      |
+-----------------+
| public:         |
|     wait();     |
|     signal();   |
+-----------------+
```

* Rather than engaging in busy waiting, the process can block itself.

*
```
typedef struct {
    int value
    struct process *list;
} semaphore

wait (semaphore *s){
    s -> value --;
    if (s -> value < 0)
        add this process to s -> list;
        block() ;
}
```

```
signal (semaphore *s){
    s->value ++;
    if (s->value <= 0){
        remove a process P from s->list ;
        wakeup (P);
    }
}
```

* Semaphore types :
    ↳ Counting (or) general semaphore : Range over an unrestricted domain
    ↳ Binary semaphore : Range only between 0 and 1.

DEADLOCK :

* Set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.

PRODUCER CONSUMER - BOUNDED BUFFER :

```
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

consumer :

```
do{
    wait (full);
    wait (mutex);

    /* remove an item from buffer to next_consumed */

    signal (mutex);
    signal (empty);
    /* consume the item in next_consumed */
} while (true);
```

Producer :

```
do{
    wait (empty);
    wait (mutex);
```

```
    /* add an item into the buffer */
    signal (mutex);
    signal (full);
  }while(true);
```

READER - WRITERS PROBLEM:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

Reader:

```
do{
    wait(mutex);
    read_count ++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);

    /* reading is performed */

    wait(mutex);
    read_count --;
    signal(mutex);
    if (read_count == 0)
        signal(rw_mutex);
}while(true);
```

WRITER:

```
do{
    wait(rw_mutex);
        /* writing is performed */
    signal(rw_mutex);
}while(true)
```

DINING PHILOSOPHERS PROBLEM:

```
semaphore chopstick[5];
do{
    wait(chopstick[i]);
    wait(chopstick[[i+1]%5]);
    /* eat for while */
    signal (chopstick[i]);
    signal (chopstick[[i+1]%5]);
    /* think for while */
}while(true);
```