

Threads: Memory and Context Switch advantages

2 INDEPENDENT PROCESSES EFFICIENT USE OF CPU OVERLAP IO & CPU

3 EXTEND TO DEPENDENT USES OF PROCESSES WITH IPC PSEUDO PARALLELISM with SHARED MEM, etc, MORE CONTEXT SWITCHES

4 MULTI-THREADING ADVANTAGES-HOW TO CONTROL?

5 USES OF THREADS IN PSEUDO PARALLEL PROGRAMMING

, Issues of implementation of Threads

6 SCHEDULING THREADS BASIC SOLUTIONS

7 SCHEDULING THREADS MIXED SOLUTIONS, NEGOTIATED IO, LWP

8 SCHEDULING THREADS MIXED SOLUTIONS, NEGOTIATED IO, LWP PICTURED

Thread programs simple examples

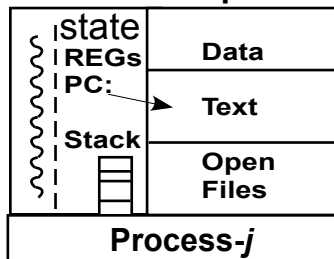
9 C MULTI-THREAD AND UNIX MULTI-PROCESS PROGRAMMING COMPARISONS

10 THREADS: PROGRAM IN C WITHOUT MUTUAL EXCLUSION

11 THREADS: PROGRAM IN C WITH SPIN LOCK MUTUAL EXCLUSION

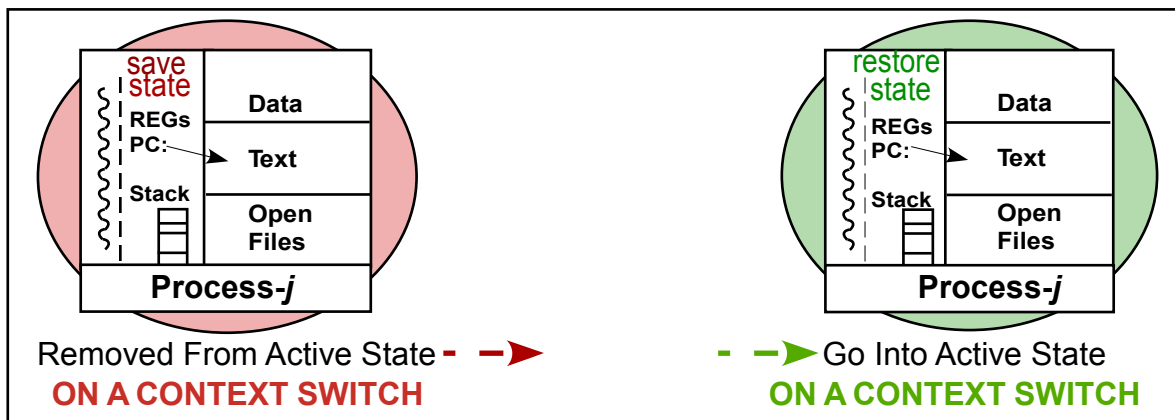
12 THREAD SCHEMA IN C WITH SEMAPHOR MUTUAL EXCLUSION

Virtual 1-CPU Independent Machine



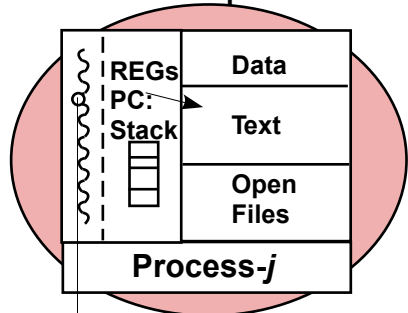
A Process Storage is partitioned into

- a) Process Wide Storage: Global Mem Data, Text, Open Files.....
- b) Dynamic Computation Path **Thread** Local Storage: States, Regs, Stack(Procedure calls)

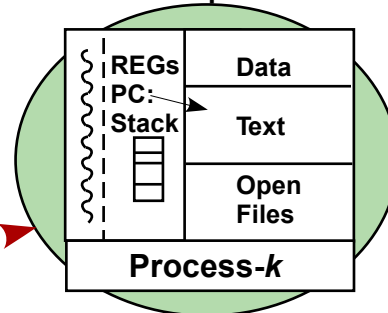


Review Highlights Of Processes

Virtual 1-CPU independent Machine



Virtual 1-CPU independent Machine



I/O, Pg Fault, Quanta

CONTEXT SWITCH

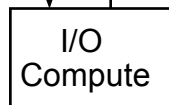
KERNEL

MULTIPROGRAMMING OVERHEAD

ON: I/O, Page Fault, Quanta...DO: SAVE Thread Position PC & REGs & Stack

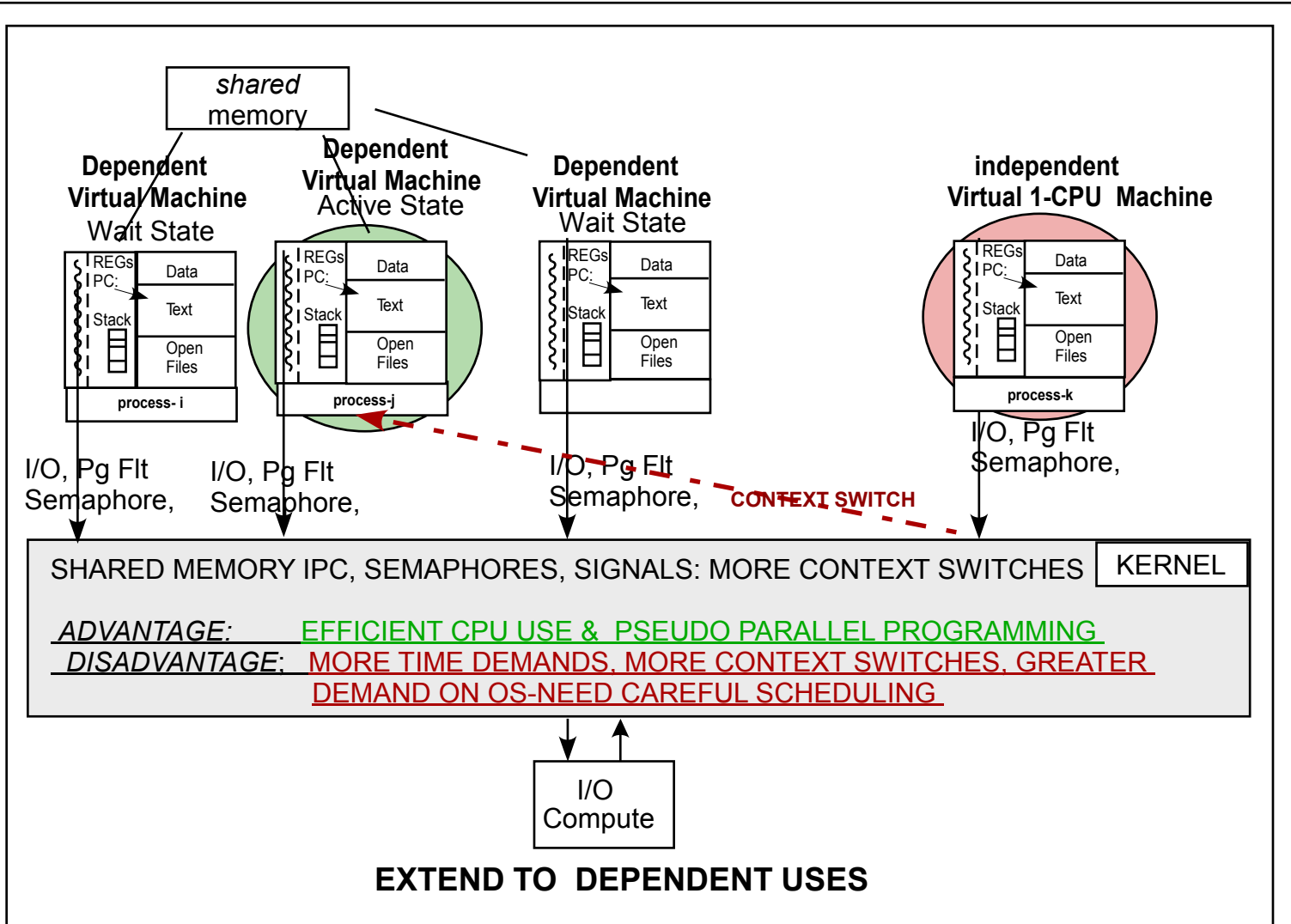
ADVANTAGE: EFFICIENT CPU USE.

DISADVANTAGE: PROCESS CONTEXT SWITCH.



1 Basic Idea- Independent Processes (No Communication)-Simple Multiprogramming

INDEPENDENT PROCESSES EFFICIENT USE OF CPU OVERLAP IO & CPU



1 More Uses-More Complexity

For Maintaining Response Time (ex. To Terminals) and For Fairness To Users We Need Quanta, Priority--Careful Scheduling (Favoring Op Sys Functions) I/O Bound,

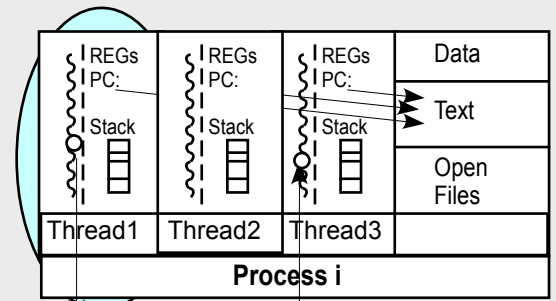
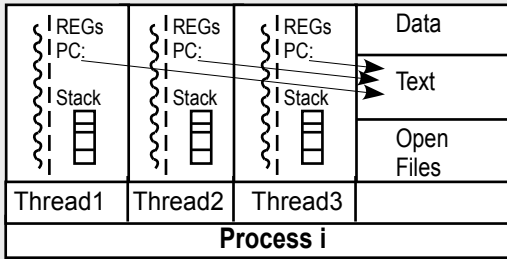
2 Pseudo Parallel: Good Way To Organize-Especially For Building The Multi-Functional Operating System., ex. A Process, Terminal, Daemons- Free List. (Many of these can remain independent but for many Applications Introduce:

Inter Process Shared Memory-Different than Intra Process local and Global Memory

2 Extend Basic Idea Dependent Processes with Communication - Simple

EXTEND TO DEPENDENT USES PROCESSES WITH IPC PSEUDO PARALLELISM with SHARED MEM, etc, MORE CONTEXT SWITCHES

Virtual Multi-CPU independent Parallel Machine



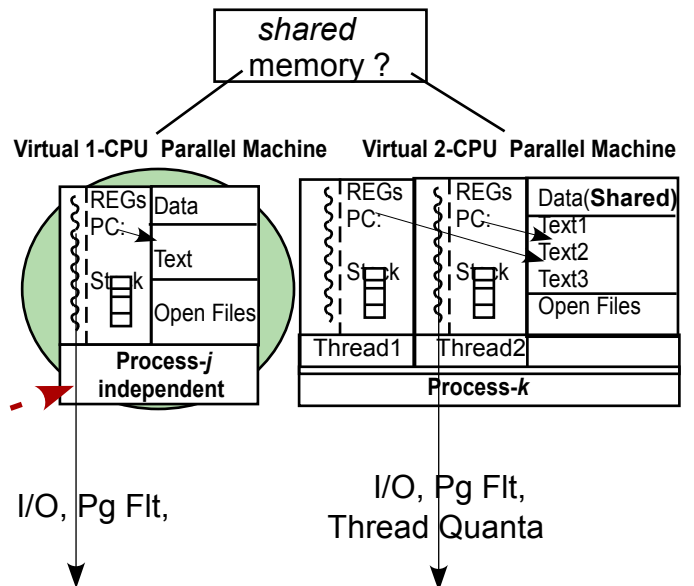
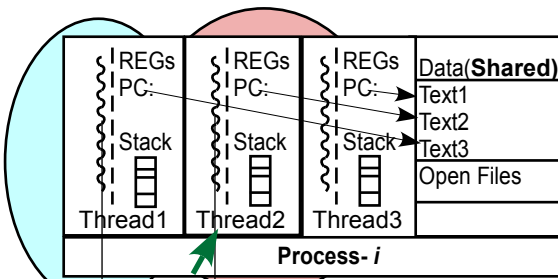
Process Storage is partitioned into

- Process Wide Storage: Global Mem Data, (Shared) Text, Open Files..
- For all Threads Dynamic Computation Path Local Storage: States, Regs, Stack(Procedure calls)

ON THREAD SWITCH
Thread1 to Thread 3-
save Thread1 restore Thread3

Virtual Parallel Machine

Virtual 3-CPU independent Parallel Machine



Thread Quanta
THREAD SWITCH
I/O, Pg Flt,

CONTEXT SWITCH

I/O, Pg Flt,

I/O, Pg Flt,
Thread Quanta

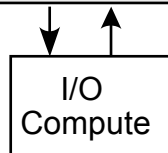
TWO TYPES OF CONTEXT SWITCH -

- PROCESS to PROCESS** Requires MUCH MEMORY and is SLOW
- THREAD to THREAD** Requires LESS MEMORY SWITCH FAST SWITCH IMPROVED EFFICIENCY

1 **ADVANTAGE:** SAVE AVERAGE TIME PER CONTEXT SWITCH. SIMPLIFY SHARING OF MEMORY. INSTEAD OF SPECIAL SHARED MEMORY OUTSIDE OF PROCESS SPACE INCORPORATE THREADS IN A SINGLE PROCESS. ALL SHARING DATA, TEXT AND OPEN FILES, AND EACH WITH ITS OWN DYNAMIC THREAD MEMORY

2 **DISADVANTAGE:** CONTROL COMPLEXITY..

KERNEL



We have seen how a number of Processes can be designed as a set of programs running in (pseudo) parallel with shared memory communication amongst them . Now, in fact, a single Process can itself be designed to consist of a number of Threads, each forming a separate program to be run in parallel. It will in fact be run in a pseudo parallel mode.

Multi Threaded Process = Each Thread a Virtual Parallel Machine”

MULTI-THREADING ADVANTAGES-HOW TO CONTROL?

Example Uses Of Multi-Threading (Each Thread May Be in A Separate Process Or All Within one Process):

Book Word Processing

- \Thread 1.,Read/ Write Terminal commands delete, scroll, changes all "to" to "two", display page x
- Thread 2. In background carrying out requested changes-may result in number of pages changing, etc
- Thread 3 automatic periodic backup to disk

Requires common access to document by all three threads--massive shared memory?

Spread Sheet

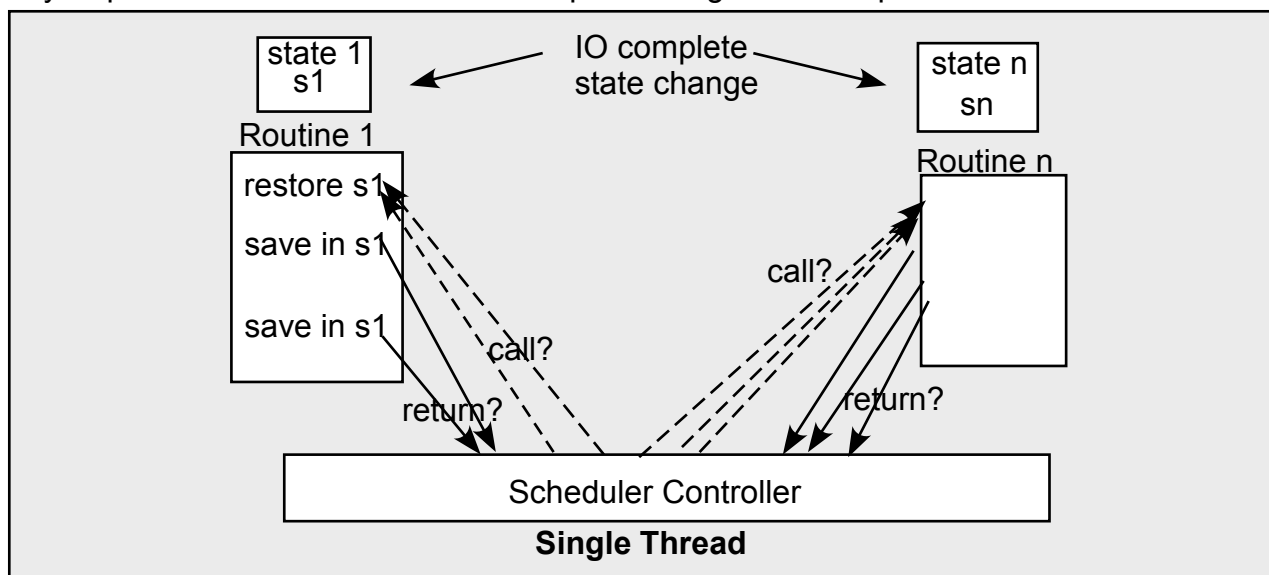
- Thread 1 Data Entry
- Thread 2 Computation based on entry.
- Thread 3 Backup

Web Server:

- Thread 1 Dispatcher: Monitors-Recieves Incoming Requests-dispatches to workers,
- Thread 2,3,..Workers which can be awakened by Dispatcher To do job requested.
check cache for material, if not there go to disk for it, blocks. (Like Shell)

Single threaded versus Multi-threaded

The choice of single threaded versus multi-threaded implementation. When a number of sub-processes which are not always able to complete immediately due to IO and/or synchronization waits, maintaining efficiency requires that wait time be used for processing other sub-processes.



In the **single threaded** case the Process must keep a record of the **state** of the various Sub-processes.. Basic Problem: A read or write , etc. takes time, and for efficiency the Process should remain active. Other parts of Process can run if Quanta is not done?. So the state of these must be known and the state of where the read was executed should be saved. All this must be handled by the application programmer in addition communication with OS to know when IO is complete is necessary.

EX: Subprocess or

- Thread 1 issues a read and blocks (State = Blocked))
- Thread 2 does a write to disk.(State = Blocked)
- Thread 3 runs.

Threads 1 and 2 can be executed only if their state has become unblocked. Therefore their state, which, is known by OS, must be communicated, and be available when thread 3 has had reasonable time.

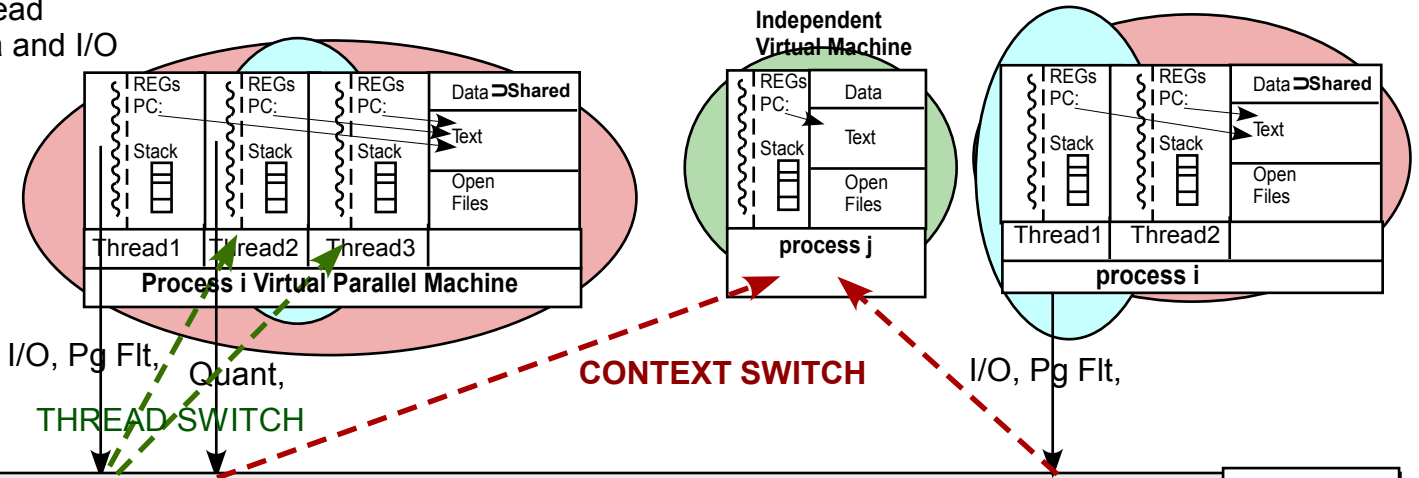
Using **multi-threading** puts the burden of the **state** maintenance and switching between activities on the multi-threading system, an OS for all Processes, or within a single Process, and so relieves the application programmer of these chores.

When threads use IO or other functions requiring waiting for an event then, for efficiency either the system takes care of multithreading or the application programmer must simulate the OS scheduling.

USES OF THREADS/PROCESSES IN PSEUDO PARALLEL PROGRAMMING

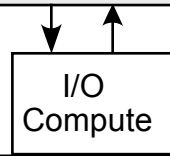
OPERATING SYSTEM HANDLES ALL (Kernel Level Threads) KLT

SAVE ON Both Thread Quanta and I/O



OPERATING SYSTEM KNOWS ABOUT ALL PROCESSES + ALL THREADS IN EACH PROCESS, IT CAN DECIDE TO DO A THREAD OR PROCESS SWITCH, ON THREAD QUANTA, PROCESS QUANTA, I/O REQUEST--SEMAPHORE, ETC.

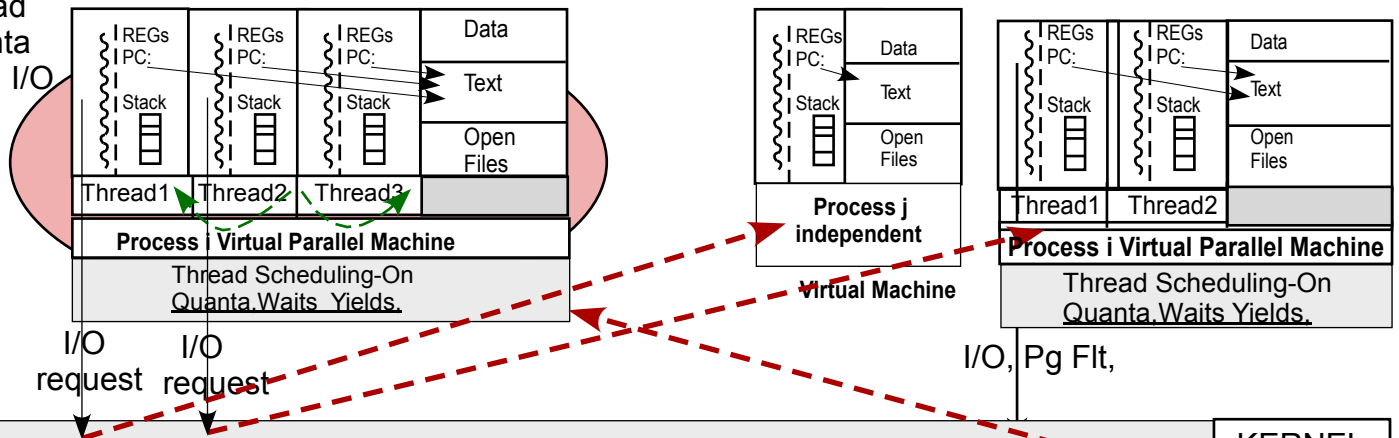
- ADVANTAGE:** AN ELABORATION OF EXISTING OP SYSTEM/ COMMON CONTROL CODE.
- DISADVANTAGE:** INCREASED COMPLICATION OF OP SYS. PROCESS NOT EASILY TRANSPORTED.



OPERATING SYSTEM HANDLES PROCESS SWITCH, THREAD SCHEDULER HANDLES THREAD SWITCHES ((User Level Threads) ULT) WITHIN PROCESS (LIBRARY ROUTINES).

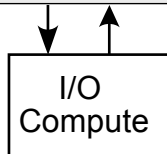
SAVE ON Thread Quanta Not on I/O

Mixed Solutions



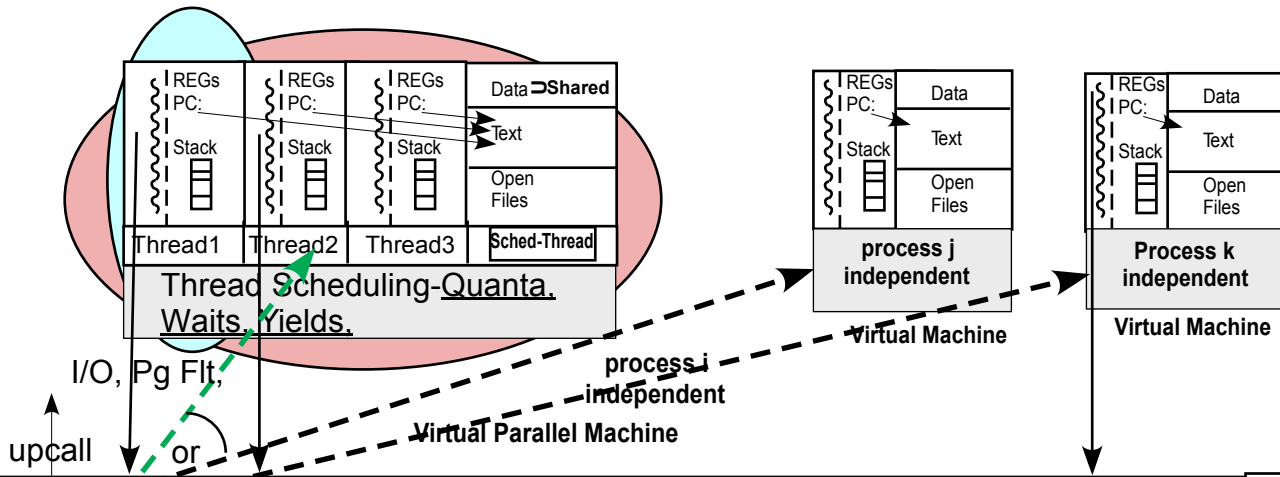
OPERATING SYSTEM KNOWS ABOUT ALL PROCESSES, NOTHING ABOUT THREADS (OTHER THAN ITS OWN). HANDLES I/O, PAGE FAULT, (EVENT REQUIRING OS- IO, MM MANAGEMENT, FILES- EACH OF WHICH REQUIRE A PROCESS SWITCH).

- ADVANTAGE:** SAVES TIME ON NON-IO SWITCHES. SIMPLE TRANSPORTABLE.
- DISADVANTAGE:** THREAD SCHEDULER PER PROCESS



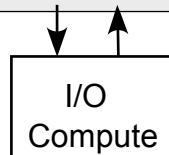
SCHEDULING THREADS BASIC SOLUTIONS

OPERATING SYSTEM HANDLES PROCESS SWITCH, When Process has had quanta But when there are I/O and page fault blocks of individual threads that could cause a switch there is Consultation With the THREAD Scheduler which HANDLES other Thread Switch WITHIN PROCESS



A Process is inoperable if all threads within the Process are blocked (doing I/O, or page fault). But one can decide, let's say, that if 2 of 5 possible threads are blocked, then the containing Process should block. An I/O call, or a page fault will go directly to the Kernel to be processed. The Kernel then informs the Thread Scheduler of these events. Then the Thread scheduler can determine if any thread is still runnable, if so run it. If not it can inform the Kernel to Block the Process in which it is contained.

ADVANTAGE: SAVES TIME ON NON-IO SWITCHES, AND LESS SO ON IO etc SCHEDULING.
DISADVANTAGE: INCREASED COMPLICATION OF PROCESS AND THREAD SCHEDULERS.



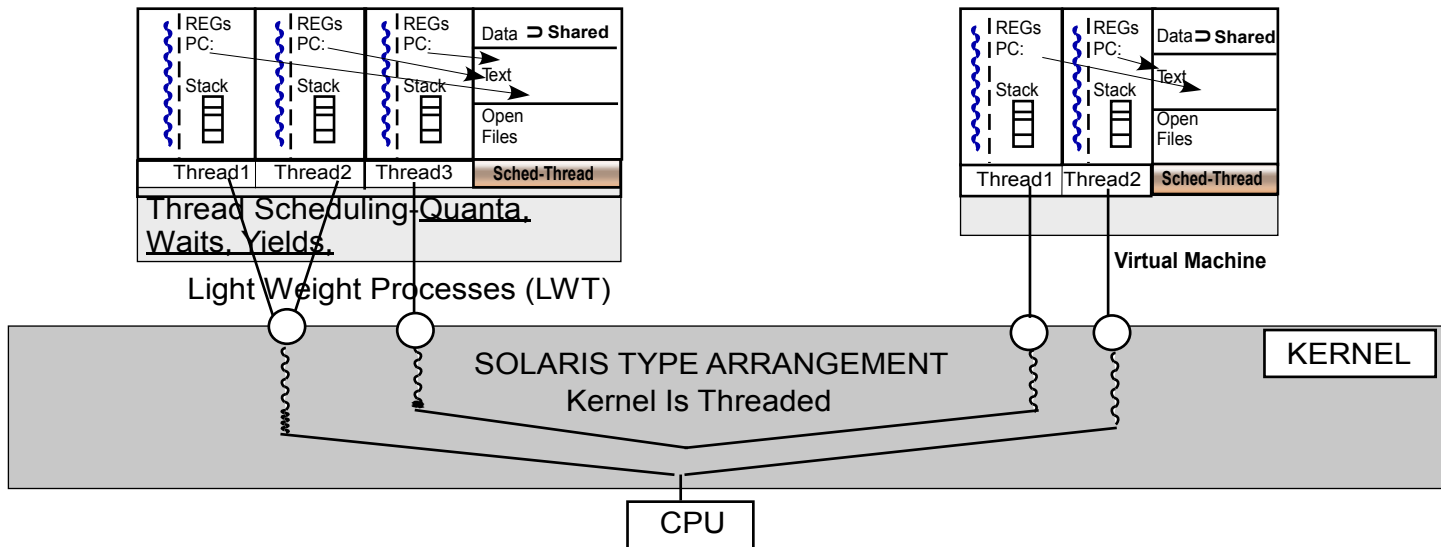
SCHEDULING THREADS MIXED SOLUTIONS, NEGOTIATED IO, LWP PICTURED

Jacket, Wrapper: Thread Scheduler: Before Issuing I/O request from Process to Kernel Check with Kernel to determine if I/O device requested is available. if so forward request to Kernel (Process Switch) if not go to a runnable thread-check for I/O available at intervals.

In thread A: read(sam/jones, Wrapper /sam/jones --> block x is block x available? **yes:** send read to kernel(context switch) **no:** run thread B check for block x periodically

Simple Extension Of User Thread Control-NEGOTIATED IO

The Thread scheduler does all scheduling,(priority, quanta, semaphore)in an Ex-Process=Task locally-(must also know about threads blocked on I/O, Pg Flt). It does all thread switches until a Process switch executed by the Kernel. When Processes became Multi-Threaded they called Tasks-the notation used here.



Each LWP is associated with one Kernel Thread and a number of the threads in a Process. It blocks threads on I/O calls/ Pg Flts with aid of its Kernel thread. Each Thread Scheduler needs to know of such blockages. When all LWPs associated with a Task are thus blocked there is a context switch executed by the Kernel.

Considerations For Maximizing Process Directed Thread Control

PCS = Process Context Switch

TCS = Thread Context Switch

LWP = Light Weight Process

1. Thread Scheduler takes care of almost everything while its in the CPU .

In order to do so: (This excludes unblockages when it is not present (can be updated) quanta)

a) It must be aware of all threads blockages including I/O PgFlt & Thread Semaphors, Mutex Waits.

i) It knows about I/O PgFlt either directly or from the Kernel

ii) Thread Semaphors, Mutexs, Waits it knows directly.

b) **All unblockages (I/O PgFault) must be revealed by the Kernel to Thread (while it is in the CPU).**

c) If all threads are blocked on I/O PgFault while the Thread Scheduler is running either:

i) Thread Scheduler knows and tells the Kernel to do a PCS. But the Kernel knows when enough are unblock to run again.

ii) The Kernel knows when all are blocked and can do a PCS.

a) This can be accomplished by knowledge of the state of all threads in each Process or

b) The Kernel can assign a LWP to a sub-group, G, of the threads in a Process. If any one in G is blocked it assumes all are blocked. Then, when all groups associated with a Process are thus found to be blocked by the LWP it can do a PCS and not again make that Process active until the last ones to be blocked are unblocked by the completion of their requested I/O, PgFl.

2. The Kernel does PCSs on Process Quanta and whenever else it sees fit to do so.

LIGHT WEIGHT PROCESSES (LWP)

SCHEDULING THREADS MIXED SOLUTIONS, NEGOTIATED IO, LWP

C MULTI-THREADING

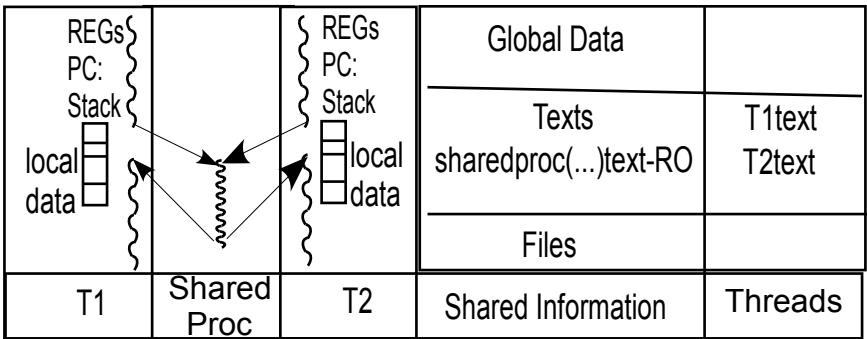
As will be illustrated in a few simple thread programs that follow, the language and consequent implementation of Multiprogramming for Threads is different than that explained previously for Processes.

1 For Processes we focussed on the **fork command**, which is the basic way in which multi-processes are formed in UNIX. Any set of Processes which are to communicate must originate from the same initial *fork* command and each new Process is an (almost) clone of the Process containing the initial *fork*.

2 Threads in UNIX do not rely on a *fork* or *fork* like implementation. The multiple Processes that are to communicate are specified effectively as **separate procedures** defined within the same program with the implication that they can be produced as separate distinct Processes, individually implemented-no cloning is implied.

Reentrant Code: Unlike C Processes, in which a Fork generated a Child Process which is a complete and separate version of the Parent Process the C-Thread code strongly suggests the possibility of *shared code*.

A procedure can be declared to be the body of more than one Process-called perhaps with different arguments. Generally the code for such a procedure should be "reentrant" or "Read Only"-that is- it should not contain any changeable instructions.



C MULTI-THREAD AND UNIX MULTI-PROCESS PROGRAMMING SOME COMPARISONS

```
void printfunct(void *ptr);
```

```
main()
```

```
{ pthread_t thread1, thread2;
```

```
char *m1 = "Hello";
```

```
char *m2 = "World";
```

```
pthread_create( &thread1,
```

```
pthread_attr_default
```

```
(void*)&printfunct, (void)* m1
```

```
);
```

```
pthread_create( &thread2,
```

```
pthread_attr_default,
```

```
(void*)&printfunct,
```

```
(void)* m2 );
```

```
);
```

```
exit(0); or sleep(0);exit(0);
```

```
}
```

```
void printfunct( void *ptr )
```

```
{
```

```
char *message;
```

```
message =(char *) ptr;
```

```
printf( " %s ", message);
```

```
}
```

```
void printfunct(void *ptr);
```

```
struct timespec delay;
```

```
main()
```

```
{ pthread_t thread1, thread2;
```

```
delay.tv_sec = 2;
```

```
char *m1 = "Hello";
```

```
char *m2 = "World";
```

```
pthread_create( &thread1,
```

```
pthread_attr_default,
```

```
(void*)&printfunct, (void)* m1
```

```
);
```

```
pthread_create( &thread2,
```

```
pthread_attr_default,
```

```
(void*)&printfunct,
```

```
(void)* m2 );
```

```
)
```

```
pthread_delay_np( &delay);
```

```
}
```

```
void printfunct( void *ptr )
```

```
{
```

```
char *message;
```

```
message =(char *) ptr;
```

```
printf( " %s ", message);
```

```
pthread_exit(0);
```

```
}
```

main is a thread-the parent thread
declare 2 names of *Thread* type

create thread with name thread1

(with minimum stack size)

function executed by thread1, called
with parameter m1

ditto for thread2

Note both threads run

the same function

This one called

with parameter m1

Trouble! Parent goes--> all threads go.,
Parent sleep --> all sleep

the function used by both-It prints the
argument it is called with.

thread time delay declaration

delay time = tv_spec_sec + tv_sec

individual thread exit

Addition

Error

Correction

THREADS: PROGRAM IN C WITHOUT MUTUAL EXCLUSION

pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex); atomic operations

Behaves like Binary Semaphore: Only 1 process can be in Critical Region at a time.

```
void producer_func(void);
void consumer_func(void);
char  buffer;                               Global (Shared By All Threads)
int   buffer_contents = 0;                  number in buffer
pthread_mutex_t mutex;                    mutex is type mutex (values 1 for open and 0 for locked)
struct timespec delay;                   delay is a struct of type timespec
main()                                     main is a thread (runs the producer)
{ pthread_t consumer;                      consumer is a thread
  delay.tv_spec_sec = 2;                   size of delays in secs + nsecs.
  delay.tv_nsec = 0;                       delay.tv_nsec = 0;
  pthread_mutex_init(&mutex, pthread_mutexattr_default); Initialize mutex = 1 not locked
                                                    no mutex attri in many versions

  pthread_create( &consumer,              create a thread
                pthread_attr_default,    thread attribute
                (void*)&consumer_func,    thread consumer executes consumer_func
                NULL                        no arguments passed to producer_func
                );
  producer_func();                          execute producer_func in thread main
}
```

```
void producer_func();
{ while(1)
  { pthread_mutex_lock( &mutex);          if mutex =0 blocked, if = 1 continue set to 0( lock)
    if (buffer_contents != N)              test buffer: not empty? if so fill
      {buffer = new_item(); buffer_contents=buffer_contents+1;}
    pthread_mutex_unlock( &mutex);        mutex = 1
    pthread_delay_np( &delay);           Delay gives consumer a chance
  }
}
```

```
void consumer_func()
{
  { while(1)
    { pthread_mutex_lock(&mutex);        if mutex =0 block, if = 1 continue and make 0 (lock)
      if (buffer_contents != 0)            test buffer: not empty
        {consume(buffer_item(); buffer_contents=buffer_contents-1} t
      pthread_mutex_unlock( &mutex);    mutex = 1
      pthread_delay_np( &delay);
    }
  }
}
```

pthread_mutex_destroy(); destroys mutexs

Note When threads sharing a lock are located on different Processors of a shared memory system then **pthread_mutex_lock(&mutex)** may be a spin lock namely if mutex is 1, it will cycle looking for mutex to change to 0 rather than be blocked by the OS. This will often be faster than the context switch involved in blocking through OS.

THREADS: PROGRAM IN C WITH SPIN LOCK MUTUAL EXCLUSION

```
void producer_func(void);
void consumer_func(void);
```

```
char  buffer;
int   buffer_contents = 0;
Semaphor producer_turn;
Semaphor consumer_turn;
```

Global (Shared By II Threads)

producer is type **Semaphor** (0-1)

```
main()
{ pthread_t consumer;
```

main is a thread (runs the producer)

```
    semaphor_init (producer_turn);
    semaphor_init (consumer_turn);
    semaphor_down(consumer_turn);
```

Semaphors initialized to 1

semaphor consumer_turn = 0, producer goes first

```
    pthread_create( &consumer,
                    pthread_attr_default,
                    (void*)&consumer_func,
                    NULL
                    );
    producer_func();
}
```

create a thread
thread attribute
thread consumer executes consumer_func
no arguments passed to producer_func

execute producer-func in thread main

```
void producer_func()
{ while(1)
  { semaphor_down( producer_turn); if 1, make 0 enter, else b locked
    {buffer = new_item();*
    (semaphor_up(consumer_turn);    increment
  }
}
```

```
void consumer_func()
{
  { while(1)
    { semaphor_down( consumer_turn);
      {buffer = new_item();*
      (semaphor_up(producer_turn);
    }
  }
}
```

Also **semaphor_destroy**(<sem-name>) , **semaphor_decrement**(<sem-name>)

* Note: This Program Schema Doesn't Include Details On How Mutual Exclusion Is Accomplished

THREAD SCHEMA IN C WITH COUNTING SEMAPHOR FOR SYNCHRONIZATION