0.0.1 Process States

- There are a number of *states* that can be attributed to a process: indeed, the operation of a multiprogramming system can be described by a state transition diagram on the process states. The states of a process include:
  - **New**–a process being created but not yet included in the pool of executable processes (*resource acquisition*).
  - **Ready**–processes that are prepared to execute when given the opportunity.
  - **Active, Running**–the process that is currently being executed by the CPU.
  - **Blocked, Waiting**–a process that cannot execute until some event occurs, such as completion of an I/O service or reception of a signal.
  - **Stopped**–a special case of **blocked** where the process is suspended by the operator or the user.
  - **Exiting, Terminated**–a process that is about to be removed from the pool of executable processes (*resource release*), a process has finished execution and is no longer a candidate for assignment to a processor, and its remaining resources and attributes are to be disassembled and returned to the operating system's "free" resource structures.
- As a process executes, it can change state due to either an external influence, e.g. it is forced to give up the CPU so that another process can take a turn, or an internal reason, e.g. it has finished or is waiting for a service from the operating system
- A process therefore takes part in a finite state system, and we typically show this in a state diagram which highlights the conditions necessary to transit from one state to another

0.1 Threads

- Process: Owner of resources allocated for individual program execution, can encompass more than one thread of execution

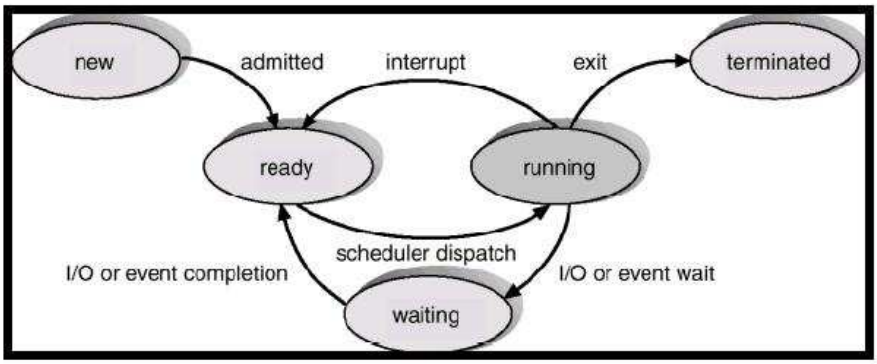

Figure 1: Diagram of Process State

- Thread: Unit of execution (unit of dispatching) and a collection of resources, with which the unit of execution is associated, characterize the notion of a process. A *thread* is the abstraction of a unit of execution. It is also referred to as a *light-weight process LWP* that share the same text (program code) and global data, but possess their own CPU register values and their own dynamic (or stack based) variables
- First look at the advantages of threads;
  - a program does not stall when one of its operations blocks.
  - save contents of a page to disk while downloading other page (for web server example)
  - Simplification of programming model
- Single process, single thread MS-DOS, old MacOS
- Single process, multiple threads OS/161
- Multiple processes, single thread traditional Unix
- Multiple processes, multiple threads modern Unices (Solaris, Linux), Windows2000
- As a basic unit of CPU utilization, a thread consists of an instruction pointer (also referred to as the PC or instruction counter), CPU register set and a stack. A thread shares its code and data, as well as system resources and other OS related information, with its peer group (other threads of the same process)

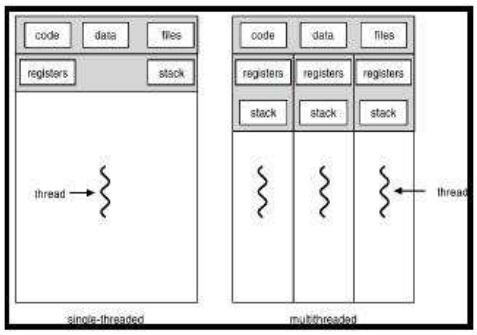

Figure 2: Single and Multithreaded Processes

- Threads versus processes;
  - A thread operates in much the same way as a process:
    - can be one of the several states.
    - executes sequentially (within a process and shares the CPU).
    - can issue system calls.
  - Economy of Overheads – managing processes is considerably more expensive than managing threads so LWPs are better.
  - Responsiveness – less setup work means faster response to requests, and multiple thread of execution mean there can be response from some threads even if other threads are busy or blocked.
  - Resource Sharing – Threads within a process share resources (including the same memory address space) conveniently and efficiently compared to separate processes
  - Threads within a process are NOT independent and are NOT protected against each other
  - Multiprocessor Use – if multiple processors are available, a multithreaded application can have its threads run in parallel which means better utilization (especially if there are few other processes present so that, without a multithreaded application, some CPUs would be idle)

- A process utilizing multithreading is a process with multiple points of execution–up to now, we have assumed that each process has only one point of execution

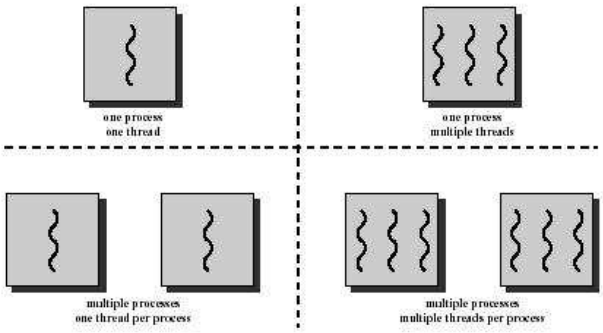

Figure 3: Threads and Processes

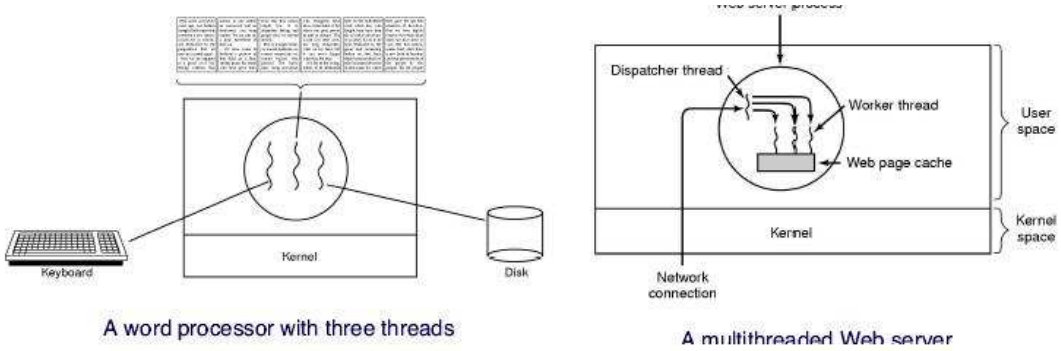

Figure 4: A word processor with three threads, a multithreaded web server

- similarity between an operating system supporting multiple processes and a process supporting multiple threads
- Figure 2 shows a traditional (or heavyweight) process, on the left, and 3 LWPs are drawn on the right in a way to emphasize their common text and global data (i.e. data and heap)
- In practice, an application such as a web server that can have considerable variations in the rate of requests can create additional threads in response to serving load, yet minimize process creation load on the host
- Less setup improves responsiveness, and shared text means more efficient memory use

0.1.1 The Thread Model

- Many-to-One
  - Many user-level threads mapped to single kernel thread.
  - Thread management is done in user space but the whole process blocks if any one user thread blocks.
  - Used on systems that do not support kernel threads.
- One-to-One
  - Each user-level thread maps to kernel thread.
  - As thread management is done in kernel space, a blocked thread does not prevent other threads from running and multiprocessor utilization is efficient.
  - Examples, Windows 95/98/NT/2000, OS/2
- Many-to-Many
  - Allows many user level threads to be mapped to many kernel threads.
  - The number of kernel threads provided might be specified according to the application and also the number of processors on a particular host.
  - Allows the operating system to create a sufficient number of kernel threads.
  - Solaris 2, DEC/compaq (Thu64), HP (HP-UX), and Silicon Graphics (IRIX), Windows NT/2000 with the *ThreadFiber* package

0.1.2 Implementing Threads in User Space

- Thread management done by user-level threads library
- a thread library is used for management with no support from (or knowledge by) the kernel. If the kernel is single threaded, and one of the user threads blocks, then the user's process is also blocked which means that the remaining user threads are also blocked. Available for many OSes
- While user threads usually emphasize their lower management load compared to kernel threads, one must consider this in relation to their lower functionality

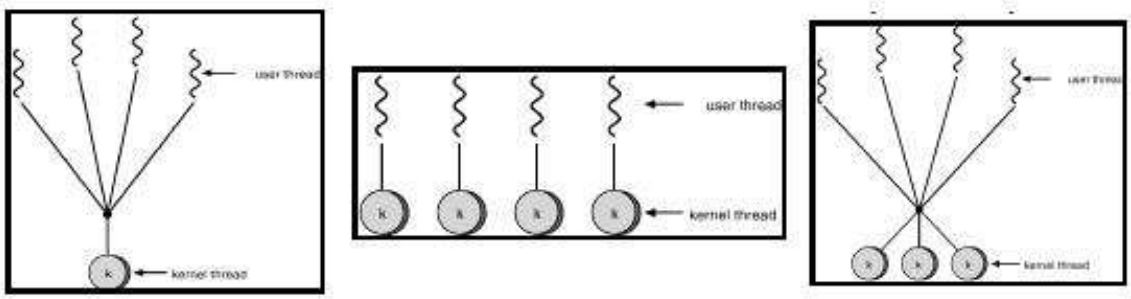

Figure 5: Thread Models; Many-to-One, One-to-One, Many-to-Many

- Examples;
  - One quite common library is *pthread* – the POSIX (POSIX is an IEEE standard for a portable operating system interface based on UNIX) thread functions.
  - Mach *C-threads*.
  - The Sun Microsystems Solaris 2 OS provides UI-*threads* (a standard originating from the Unix International Organization, RIP).

0.1.3 Implementing Threads in the Kernel

- Supported by the kernel, the kernel performs all management (creation, scheduling, deletion, etc.)
- if one thread blocks, another may be run
- If the kernel is managing multiple processors, an efficient mapping of threads to processors is possible
- Examples; Windows 95/98/NT/2000, Solaris, Tru64 UNIX, BeOS, Linux

0.2 Interprocess Communication

- Multi-threading: concurrent threads share an address space
- Multi-programming: concurrent processes execute on a uniprocessor
- Multi-processing: concurrent processes on a multiprocessor
- Distributed processing: concurrent processes executing on multiple nodes connected by a network

- Concurrent processes (threads) need special support:
  - Communication among processes
  - Allocation of processor time
  - Sharing of resources
  - Synchronization of multiple processes
- In a multiprogramming environment, processes executing concurrently are either *competing* for the CPU and other global resources, or *cooperating* with each other for sharing some resources
- An OS deals with competing processes by carefully allocating resources and properly isolating processes from each other. For cooperating processes, on the other hand, the OS provides mechanisms to share some resources in certain ways as well as allowing processes to properly interact with each other
- Cooperation is either by implicit sharing or by explicit communication
- Processes: *competing* Processes that do not exchange information cannot affect the execution of each other, but they can compete for devices and other resources. Such processes do not intend to work together, and so are unaware of one another
- Properties: Deterministic, Reproducible,Can stop and restart without "side" effects, Can proceed at arbitrary rate
- Processes: *cooperating* Processes that are aware of each other, and directly (by exchanging messages) or indirectly (by sharing a common object) work together, may affect the execution of each other
- Properties: Share (or exchange) something: a common object (or a message), Non-deterministic (a problem!), May be irreproducible (a problem!), Subject to race conditions (a problem!)
- Threads of a process usually do not compete, but cooperate
- Why cooperation? We allow processes to cooperate with each other, because we want to:
  - share some resources.
  - do things faster
    - Read next block while processing current one.

Table 1: Race Condition

Process A	**Process B**	**concurrent access**
A = 1;	**B = 2;**	*does not matter*
A = B + 1;	**B = B * 2;**	*important!*

    - Divide jobs into smaller pieces and execute them concurrently.
  - construct systems in modular fashion.
  - UNIX example:

```
cat infile | tr ' ' '\012' |tr '[A-Z]' '[a-z]' | sort | uniq -c
```

0.2.1 Race Conditions

- A potential problem; Instructions of cooperating processes can be interleaved arbitrarily. Hence, the order of (some) instructions are irrelevant. However, certain instruction combinations must be eliminated. For example: see Table 1

- A *race condition* is a situation where two or more processes access shared data concurrently and correctness depends on specific interleavings of operations; final value of shared data depends on *timing* (i.e., *race* to access and modify data)

- To prevent race conditions, concurrent processes must be **synchronized**

0.2.2 Critical Regions

- A *section of code*, or a *collection of operations*, in which only one process may be executing at a given time and which we want to make "sort of" atomic. *Atomic* means either an operation happens in its entirely (everything happens at once) or NOT at all; i.e., it cannot be interrupted in the middle. Atomic operations are used to ensure that cooperating processes execute correctly. Mutual exclusion mechanisms are used to solve the *critical region* problem

- machine instructions are atomic, high level instructions are not (count++; this is actually 3 machine level instructions, an interrupt can occur in the middle of instructions)

- Fundamental requirements; Concurrent processes should meet the following requirements in order to cooperate correctly and efficiently using shared data:
  - *Mutual exclusion*–no two processes will simultaneously be inside the same critical region (CR).
  - *No assumptions*–may be made about speeds or the number of CPUs. Must handle all possible interleavings.
  - *Fault tolerance*–processes running outside their CR should not block with others accessing the CR.
  - *Progress*–no process should have to wait forever to enter its CR. A process wishing to enter its CR will eventually do so in finite time.

  Also, a process in one CR should not block others entering a different CR. *Efficiency*–a process will remain inside its CR for a short time only, without blocking.
- Conceptually, there are three ways to satisfy the implementation requirements:
  - Software approach: put responsibility on the processes themselves
  - Systems approach: provide support within operation system or programming language
  - Hardware approach: special-purpose machine instructions

0.2.3 Mutual Exclusion with Busy Waiting (Software approach)

- *Mutual exclusion* is a mechanism to ensure that only one process (or person) is doing certain things at one time, thus avoid data inconsistency. All others should be prevented from modifying shared data until the current process finishes
- Strict Alternation (see Fig. 7)
  - the two processes strictly alternate in entering their CR
  - the integer variable **turn,** initially 0, keeps track of whose turn is to enter the critical region
  - **busy waiting**, continuously testing a variable until some value appears, a lock that uses busy waiting is called a **spin lock**

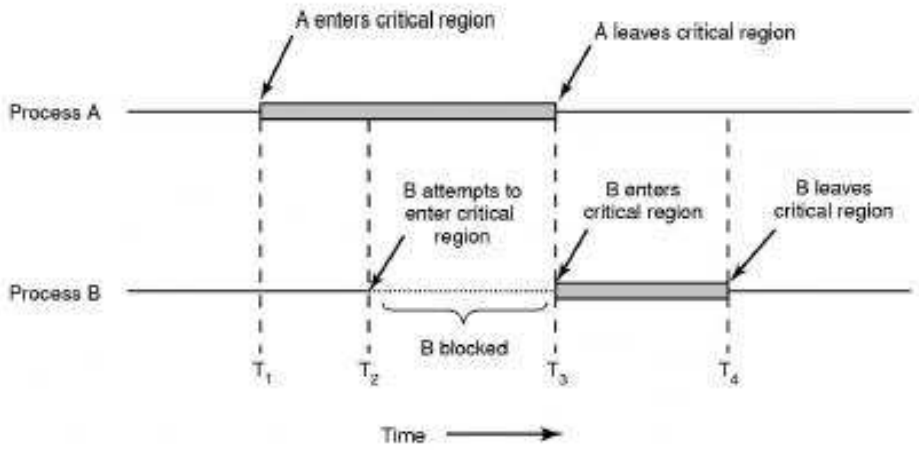

Figure 6: Mutual exclusion using critical regions

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Figure 7: A proposed solution to the CR problem. (a) Process 0, (b) Process 1

  - both processes are executing in their noncritical regions
  - process 0 finishes its noncritical region and goes back to the top of its loop
  - unfortunately, it is not permitted to enter its CR, **turn** is 1 and process 1 is busy with its nonCR
  - this algorithm does avoid all races
  - but violates condition 3

- Petersons's solution (see Fig. 8)
  - does not require strict alternation

- this algorithm consists of two procedures
- before entering its CR, each process calls **enter_region** with its own process number, 0 or 1
- after it has finished with the shared variables, the process calls **leave_region** to allow the other process to enter
- consider the case that both processes call **enter_region** almost simultaneously
- both will store their process number in **turn .** Whichever store is done last is the one that counts; the first one is overwritten and lost
- suppose that process 1 stores last , so **turn** is 1.
- when both processes come to the **while** statement, process 0 enters its critical region
- process 1 loops until process 0 exists its CR
- no violation, implements mutual exclusion
- burns CPU cycles (requires busy waiting), can be extended to work for n processes, but overhead, cannot be extended to work for an unknown number of processes, unexpected effects (i.e.,**priority inversion problem**)

0.2.4 Sleep and wakeup

- blocks instead of wasting CPU time (while loop) when they are not allowed to enter their CRs
- *sleep* and *wakeup* pair
- *sleep* is a system call that causes the caller to block (be suspended until another process wakes is up)
- The Producer-Consumer Problem
  - Suppose one process is creating information that is going to be used by another process, e.g., suppose one process reads information from the disk, and another compiles that information from source to machine code.
  - Producer: creates copies of a resource
  - Consumer: uses up copies of a resource

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 - process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Figure 8: Peterson' solution for achieving mutual exclusion

- Buffers: used to hold information after producer has created it but before consumer has used it
- Signaling: keeping control of producer and consumer (e.g., preventing overrun of the producer)
- Constraints:
  - Consumer must wait for a producer to fill buffers. (signaling)
  - Producer must wait for consumer to empty buffers, when all buffer space is in use. (signaling)
  - Only one process must manipulate buffer pool at once. (mutual exclusion)
- Trouble arises when the producer wants to put a new item in the buffer, but it is already full
- The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items
- RACE CONDITION can occur because access to *count* (see Fig. 9) is unconstrained.
  - the buffer is empty
  - the consumer has read *count* to see if it is 0, **sleeping**

* at that instant, the scheduler started running the producer
* the producer inserts an item in the buffer, *count* is 1
* the consumer should be awaken up, the producer calls **wakeup**
* the consumer is not logically asleep, so the **wakeup** signal is lost
* the producer will fill up the buffer and also go to sleep
* BOTH WILL SLEEP FOREVER.

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item();                /* generate next item */
          if (count == N) sleep();              /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep();              /* if buffer is empty, got to sleep */
          item = remove_item();                 /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Figure 9: The producer-consumer problem with a fatal race problem