Multithreaded Programming with POSIX Pthreads

Processes Revisited

- A process is an active runtime environment that accommodates a running program, providing an execution state along with certain resources, including file handles and registers, along with:
  - a program counter (Instruction Pointer)
  - a process id, a process group id, etc.
  - a process stack
  - one or more data segments
  - a heap for dynamic memory allocation
  - a process state (running, ready, waiting, etc.)
- Informally, a process is an executing program

Multiprocessing Revisited

- A multiprocessing or multitasking operating system (like Unix, as opposed to DOS) can have more than one process executing at any given time
- This simultaneous execution may either be
  - concurrent, meaning that multiple processes in a run state can be swapped in and out by the OS
  - parallel, meaning that multiple processes are actually running at the same time on multiple processors

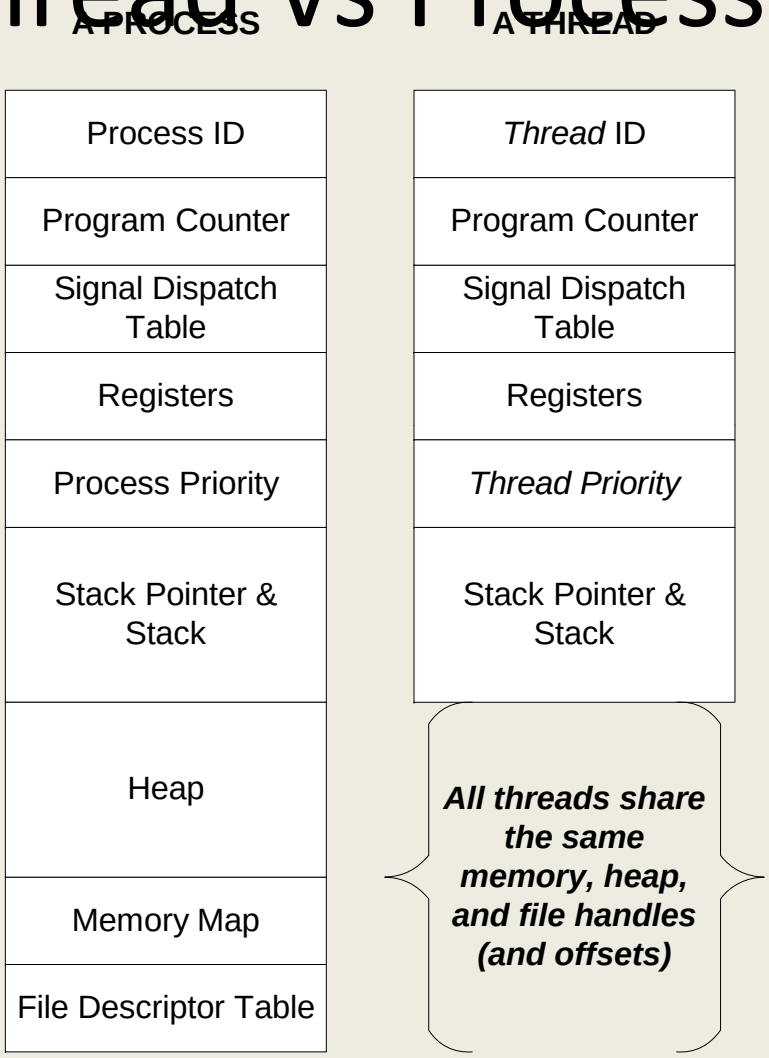
What is a Thread?

- A thread is an encapsulation of some flow of control in a program, that can be independently scheduled
- Each process is given a single thread by default
- A thread is sometimes called a lightweight process, because it is similar to a process in that it has its own thread id, stack, stack pointer, a signal mask, program counter, registers, etc.

What is a Thread?

- All threads within a given process share resource handles, memory segments (heap and data segments), and code. THEREFORE :
  - All threads share the same data segments and code segments

Thread Vs Process



Processes Vs Threads: Creation Times

- Because threads are by definition lightweight, they can be created more quickly that “heavy” processes:
  - Sun Ultra5, 320 Meg Ram, 1 CPU
    - 94 forks()/second
    - 1,737 threads/second (18x faster)
  - Sun Sparc Ultra 1, 256 Meg Ram , 1 CPU
    - 67 forks()/second
    - 1,359 threads/second (20x faster)
  - Sun Enterprise 420R, 5 Gig Ram, 4 CPUs
    - 146 forks()/second
    - 35,640 threads/second (244x faster)
  - Linux 2.4 Kernel, .5 Gig Ram, 2 CPUs
    - 1,811 forks()/second
    - 227,611 threads/second (125x faster)

Processes Vs Threads

- Threads can be created and managed more quickly than processes because:
  - Threads have less overhead than processes, for example, threads share the process heap, all data and code segments
  - Threads can live entirely in user space, so that no kernel mode switch needs to be made to create a new thread
  - Processes don’t need to be swapped to create a thread

Analogies

- Just as a multitasking operating system can have multiple processes executing concurrently or in parallel, so a single process can have multiple threads that are executing concurrently or in parallel
- These multiple threads can be task swapped by a scheduler onto a single processor (via a LWP), or can run in parallel on separate processors

Benefits of Multithreading

- Performance gains
  - Amdahl's Law: speedup = 1 / ((1 – p) + (p/n))
  - the speedup generated from parallelizing code is the time executing the parallelizable work (p) divided by the number of processors (n) plus 1 minus the parallelizable work (1-p)
  - The more code that can run in parallel, the faster the overall program will run
  - If you can apply multiple processors for 75% of your program's execution time, and you're running on a dual processor box:
    - 1 / ((1 - .75) + (.75 / 2)) = 60% improvement
  - Why is it not strictly linear? How do you calculate p?

Benefits of Multithreading (continued)

- Increased throughput
- Increased application responsiveness
- Replacing interprocess communications (you're in one process)
- Single binary executable runs on both multiprocessors as well as single processors (processor transparency)
- Gains can be seen even on single processor machines, because blocking calls no longer have to stop you.

What is POSIX Thread?

- Each OS had it's own thread library and style
- That made writing multithreaded programs difficult because:
  - you had to learn a new API with each new OS
  - you had to modify your code with each port to a new OS
- POSIX (IEEE 1003.1c-1995) provided a standard known as Pthreads
- DCE threads were based on an early 4th draft of the POSIX Pthreads standard (immature)
- Unix International (UI) threads (Solaris threads) are available on Solaris (which also supports POSIX threads)

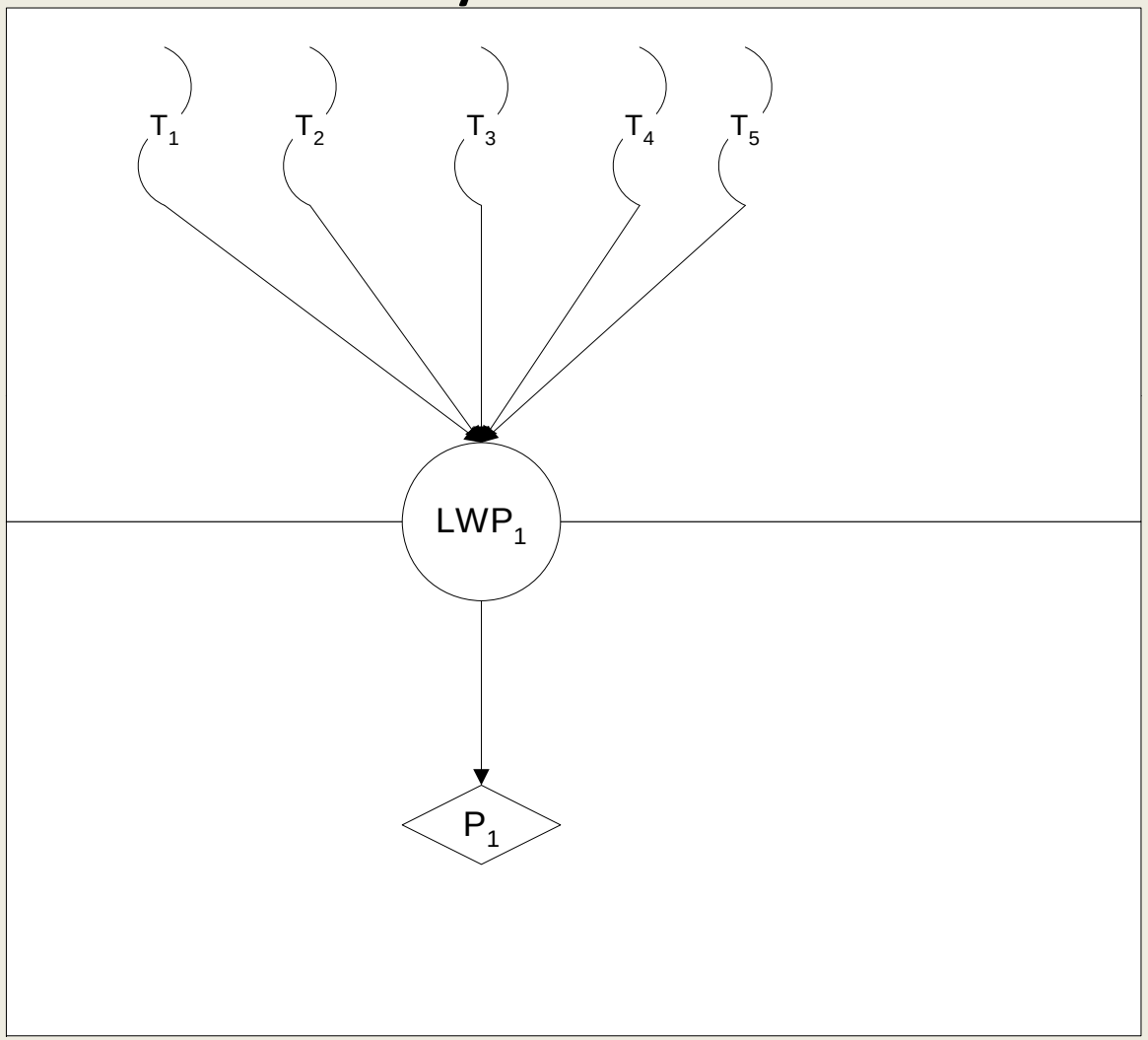
On the Scheduling of Threads

- Threads may be scheduled by the system scheduler (OS) or by a scheduler in the thread library (depending on the threading model).
- The scheduler in the thread library:
  - will preempt currently running threads on the basis of priority
  - does NOT time-slice (i.e., is not fair). A running thread will continue to run forever unless:
    - a thread call is made into the thread library
    - a blocking call is made
    - the running thread calls sched_yield()

Models

- Many Threads to One LWP
  - DCE threads on HPUX 10.20
- One Thread to One LWP
  - Windows NT
  - Linux (clone() function)
- Many Threads to Many LWPs
  - Solaris, Digital UNIX, IRIX, HPUX 11.0)

Many Threads to One LWP



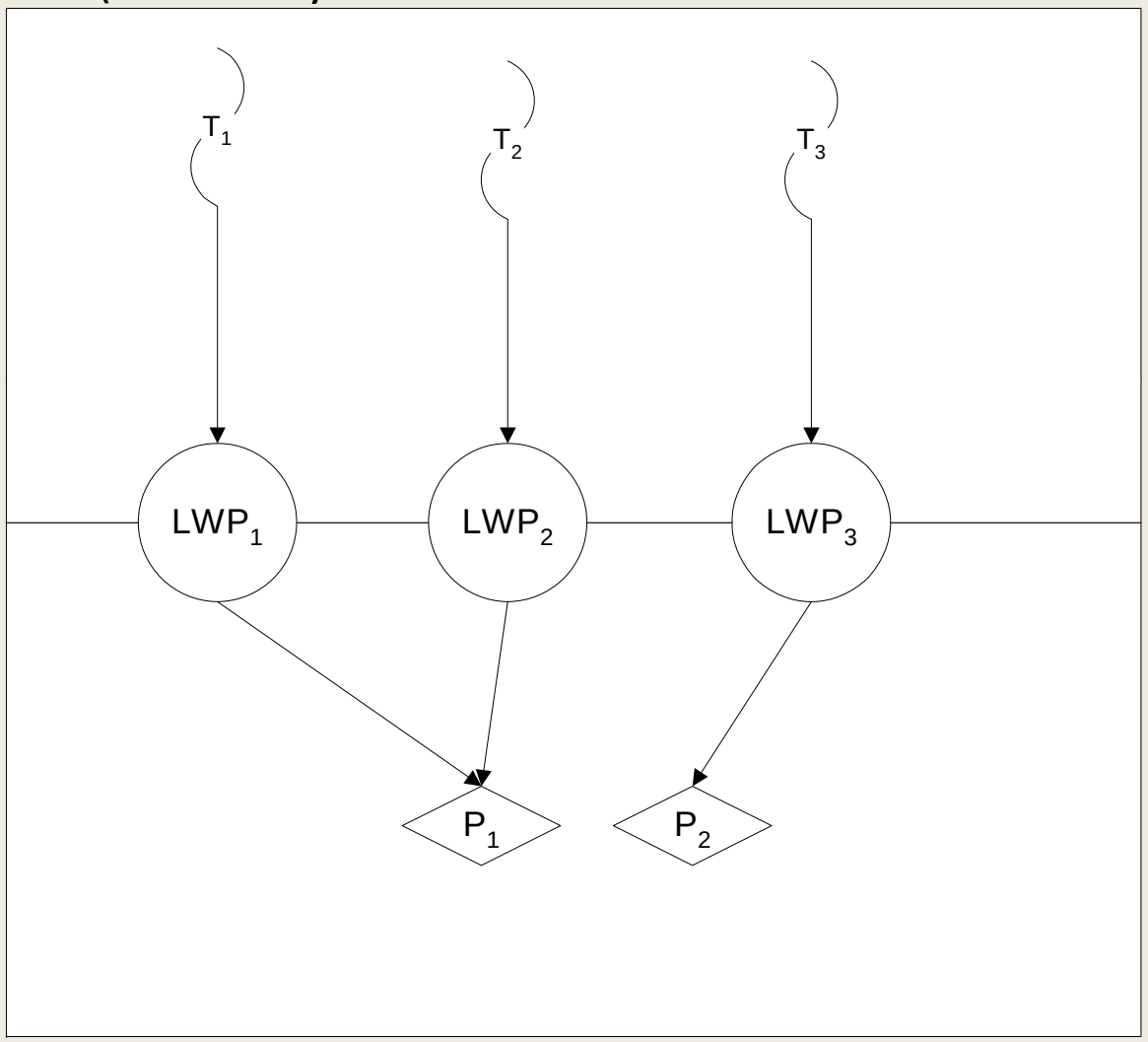
USER SPACE

AKA "user space threads". All threads are "invisible" to the kernel (therefore cannot be schdeduled individually by the kernel). Since there's only a single LWP (kernel-scheduled entity), user space threads are multiplexed onto a single processor. The kernel sees this process as "single threaded" because it only sees a single LWP.

KERNEL SPACE

- very fast context switches between threads is executed entirely in user space by the threads library
- unlimited number of user threads (memory limit) can support logical concurrency model only
- parallelism is not possible, because all user threads map to a single kernel-schedulable entity (LWP), which can only be mapped on to a single processor
- Since the kernel sees only a single process, when one user space thread blocks, the entire process is blocked, effectively block all other user threads in the process as well

One Thread to One LWP(Windows NT, Linux)
(there may be no real distinction between a thread and LWP)



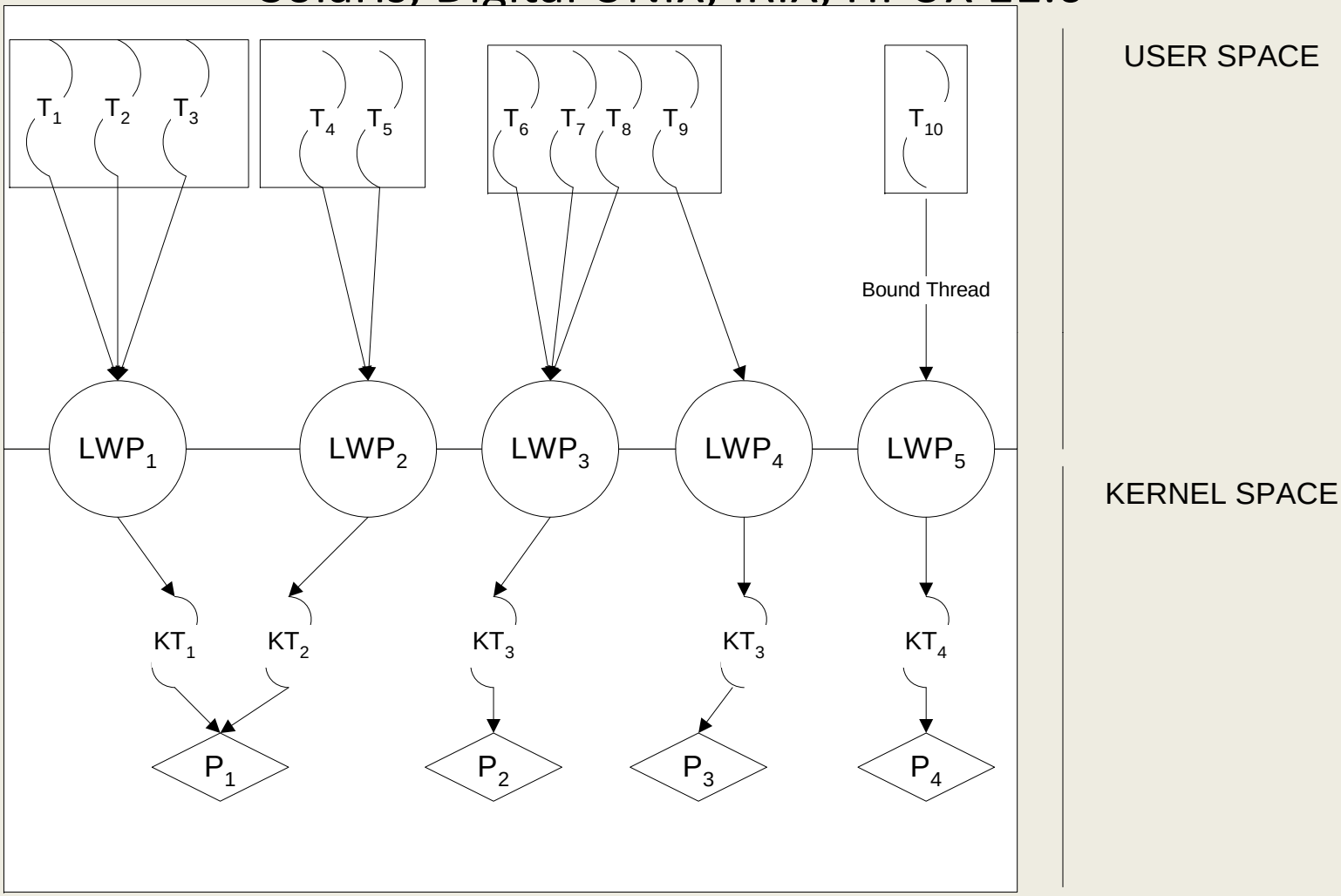
USER SPACE
Each user space thread is associated with a single kernel thread to which it is permanently bound. Because each user thread is essentially a kernel-schedulable entity, parallel execution is supported.

KERNEL SPACE
The 1x1 model executes in kernel space, and is sometimes called the Kernel Threads model. The kernel selects kernel threads to run, and each process may have one or more threads

1x1 Model Variances

- Parallel execution is supported, as each user thread is directly associated with a single kernel thread which is scheduled by the OS scheduler
- slower context switches, as kernel is involved
- number of threads is limited because each user thread is directly associated with a single kernel thread (in some instances threads take up an entry in the process table)
- scheduling of threads is handled by the OS's scheduler, threads are seldom starved
- Because threads are essentially kernel entities, swapping involves the kernel and is less efficient than a pure user-space scheduler

Many Threads to Many LWPs
Solaris, Digital UNIX, IRIX, HPUX 11.0



MxN Model Variances

- Extraordinarily flexible, bound threads can be used to handle important events, like a mouse handler
- Parallel execution is fully supported
- Implemented in both user and kernel space
- Slower context switches, as kernel is often involved
- Number of user threads is virtually unlimited (by available memory)
- Scheduling of threads is handled by both the kernel scheduler (for LWPs) and a user space scheduler (for user threads). User threads can be starved as the thread library's scheduler does not preempt threads of equal priority (not RR)
- The kernel sees LWPs. It does NOT see threads

Creating a POSIX Thread:

- #include <pthread.h>
- void * pthread_create(pthread_t *thread, const pthread_attr_t attr, void *(*thrfunc)(void *), void *args);
- Each thread is represented by an identifier, of type pthread_t
- Code is encapsulated in a thread by creating a thread function (cf. “signal handlers”)
- Attributes may be set on a thread (priority, etc.). Can be set to NULL.
- An argument may be passed to the thread function as a void **

Creation of a Thread: An Example

```
void *thread_routine(void * arg)
{
	return arg;
}

main(int argc , char *argv[])
{
	pthread_t = tid;
	void * tResult;
	int status;
	status = pthread_create(&tid, NULL, thread_routine, NULL);
	status = pthread_join(tid, &tResult);
}
```

Life Cycle of a Thread

- Ready
- Running
- Blocked
- Terminated

Detaching a Thread

- int pthread_detach(pthread_t threadid);
- Detach a thread when you want to inform the operating system that the threads return result is unneeded
- Detaching a thread tells the system that the thread (including its resources—like a 1Meg default stack on Solaris!) is no longer being used, and can be recycled
- A detached thread's thread ID is undetermined.
- Threads are detached after a pthread_detach() call, after a pthread_join() call, and if a thread terminates and the PTHREAD_CREATE_DETACHED attribute was set on creation

“Wating” on a Thread

- int pthread_join(pthread_t thread, void** retval);
- pthread_join() is a blocking call on non-detached threads
- It indicates that the caller wishes to block until the thread being joined exits
- You cannot join on a detached thread, only non-detached threads (detaching means you are NOT interested in knowing about the threads exit)

Exiting from a Thread Function

- int pthread_exit(void * retval);
- A thread ends when it returns from (falls out of) its thread function encapsulation
- A detached thread that ends will immediately relinquish its resources to the OS
- A non-detached thread that exists will release some resources but the thread id and exit status will hang around in a zombie-like state until some other thread requests its exit status via pthread_join()

Miscellaneous Functions

- pthread_t pthread_self(void);
  - pthread_self() returns the currently executing thread's ID
- int sched_yield(void);
  - sched_yield() politely informs the thread scheduler that your thread will willingly release the processor if any thread of equal or lower priority is waiting

Miscellaneous Functions

- int pthread_setconcurrency(int threads);
  - pthread_setconcurrency() allows the process to request a fixed minimum number of light weight processes to be allocated for the process. This can, in some architectures, allow for more efficient scheduling of threads