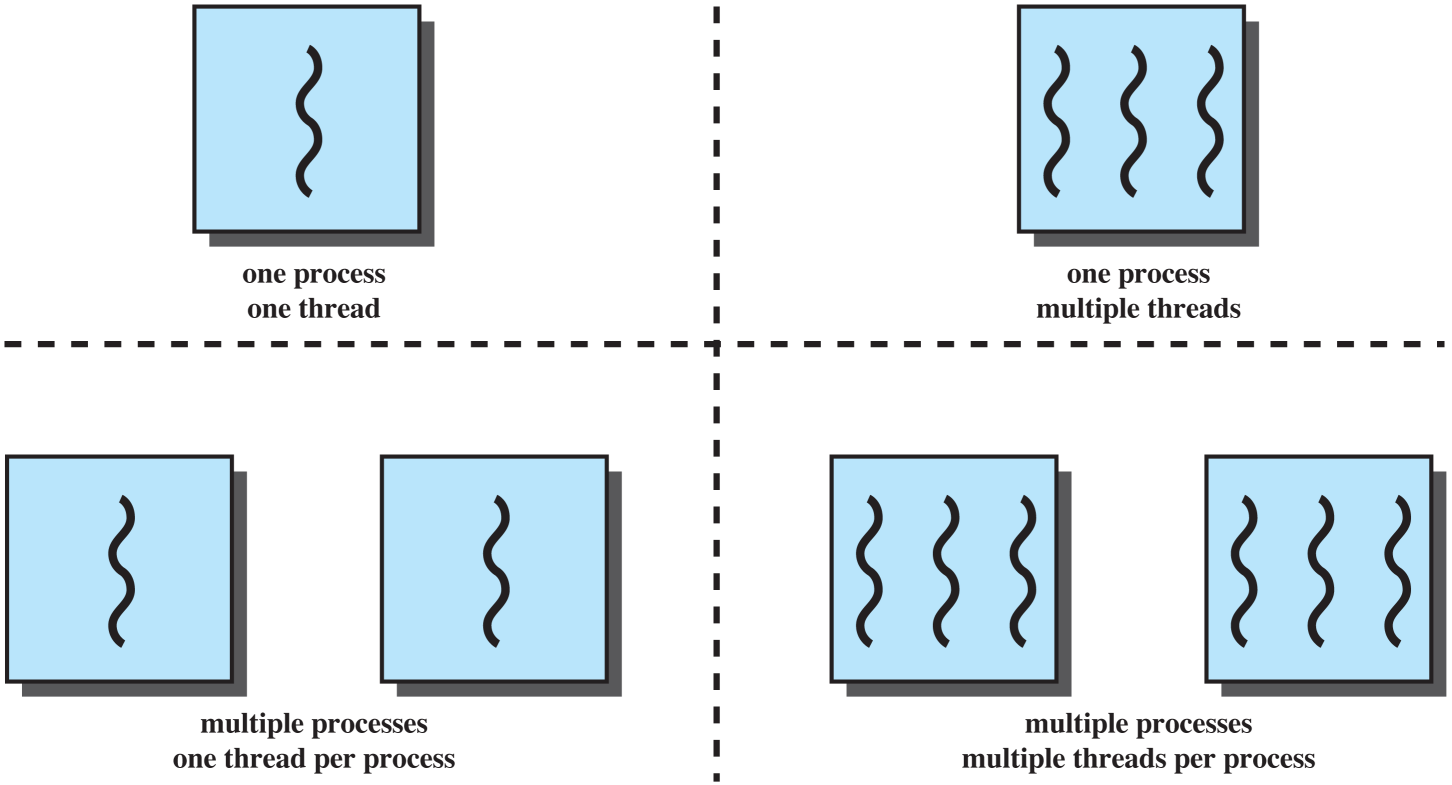
PROCESSES AND THREADS

- Process characteristics:
  - Resource Ownership (virtual address space including the process image)
    - the OS performs a protection function to prevent unwanted interference between processes with respect to resources
  - Scheduling/Execution (follows an execution path that may be interleaved with other processes)
    - a process has an execution state (Running, Ready, etc.) and a dispatching priority and is scheduled and dispatched by the OS
    - Can be separated from resource ownership
- The unit of dispatching is referred to as a thread or lightweight process
- The unit of resource ownership is referred to as a process or task
- Multithreading = The ability of an OS to support multiple, concurrent paths of execution within a single process

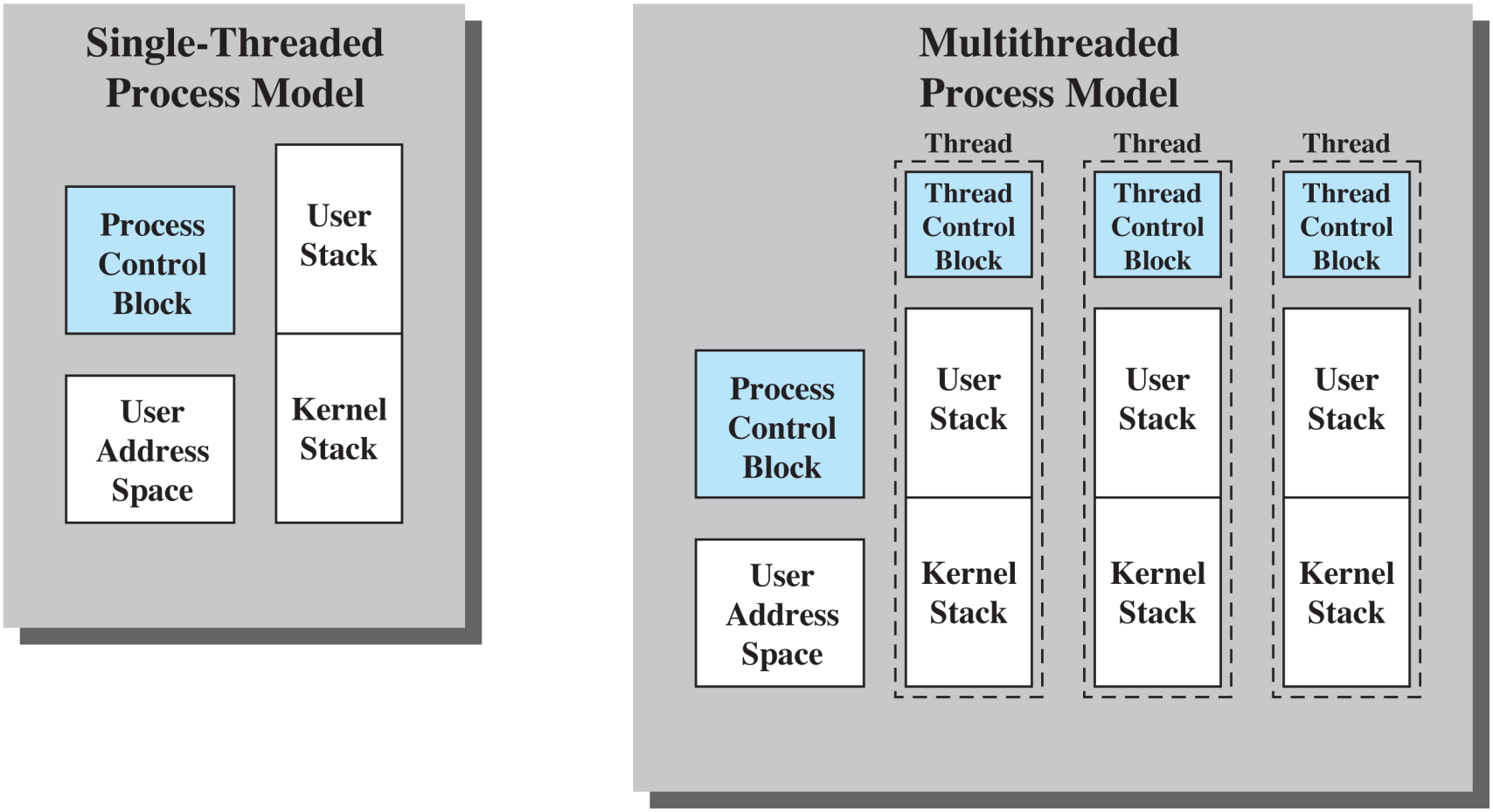
PROCESS/THREAD APPROACHES



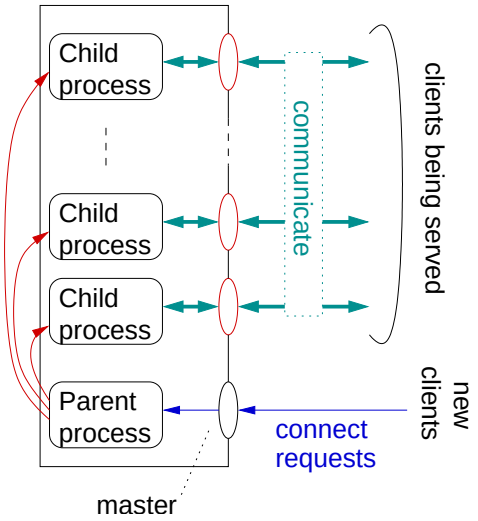
- Example: DOS (“single-threaded”), Java VM (single process, multiple threads)

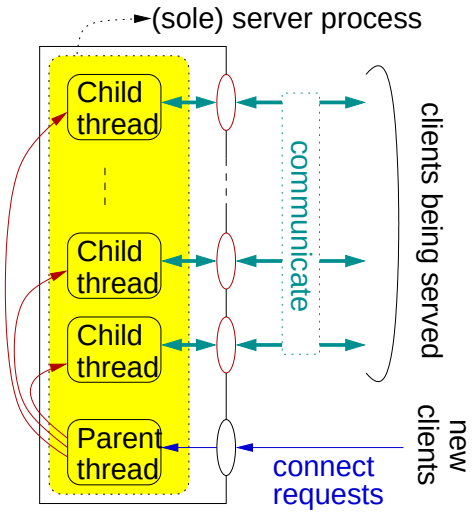
THREADS IN PROCESSES

- The process is a unit of resource allocation and resource protection
  - A virtual address space that holds the process image
  - Protected access to: CPU, other processes, files, I/O resources
  - Contains multiple threads of execution
- Each thread has:
  - An execution state (Running, Ready, etc.)
  - Saved thread context when not running
  - An execution stack
  - Some per-thread static storage for local variables
  - Access to memory and resources of a process (shared by all threads in process)
- Thread synchronization
  - Necessary to synchronize the activities of the various threads in a process
  - All threads of a process share the same address space and other resources
  - Alteration of a resource by one thread affects the other threads in the process


Single-Threaded Process Model
Process Control Block
User Stack
User Address Space
Kernel Stack
Multithreaded Process Model
Thread
Thread
Thread
Thread Control Block
Thread Control Block
Thread Control Block
Process Control Block
User Stack
User Stack
User Stack
User Address Space
Kernel Stack
Kernel Stack
Kernel Stack

THREADS VS PROCESSES EXAMPLE: NETWORK SERVERS





Repeat forever:

1. Accept connection request, create a new slave socket s, and fork
2. If children process then
   (a) Close master socket
   (b) Interact with client through s
3. Otherwise (i.e., if parent process):
   (a) Close slave socket

Repeat forever:

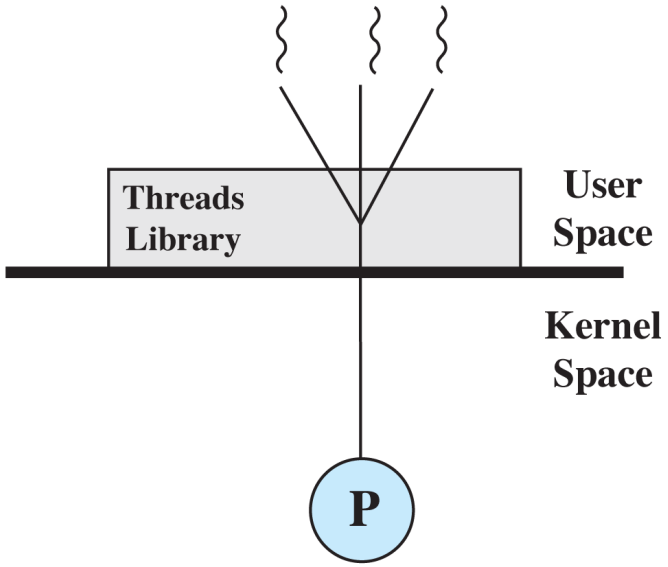
1. Accept connection request, create a new slave socket s, and create new thread
2. In the new thread:
   (a) Do not close master socket
   (b) Interact with client through s
3. In the main thread:
   (a) Do not close slave socket

THREADS

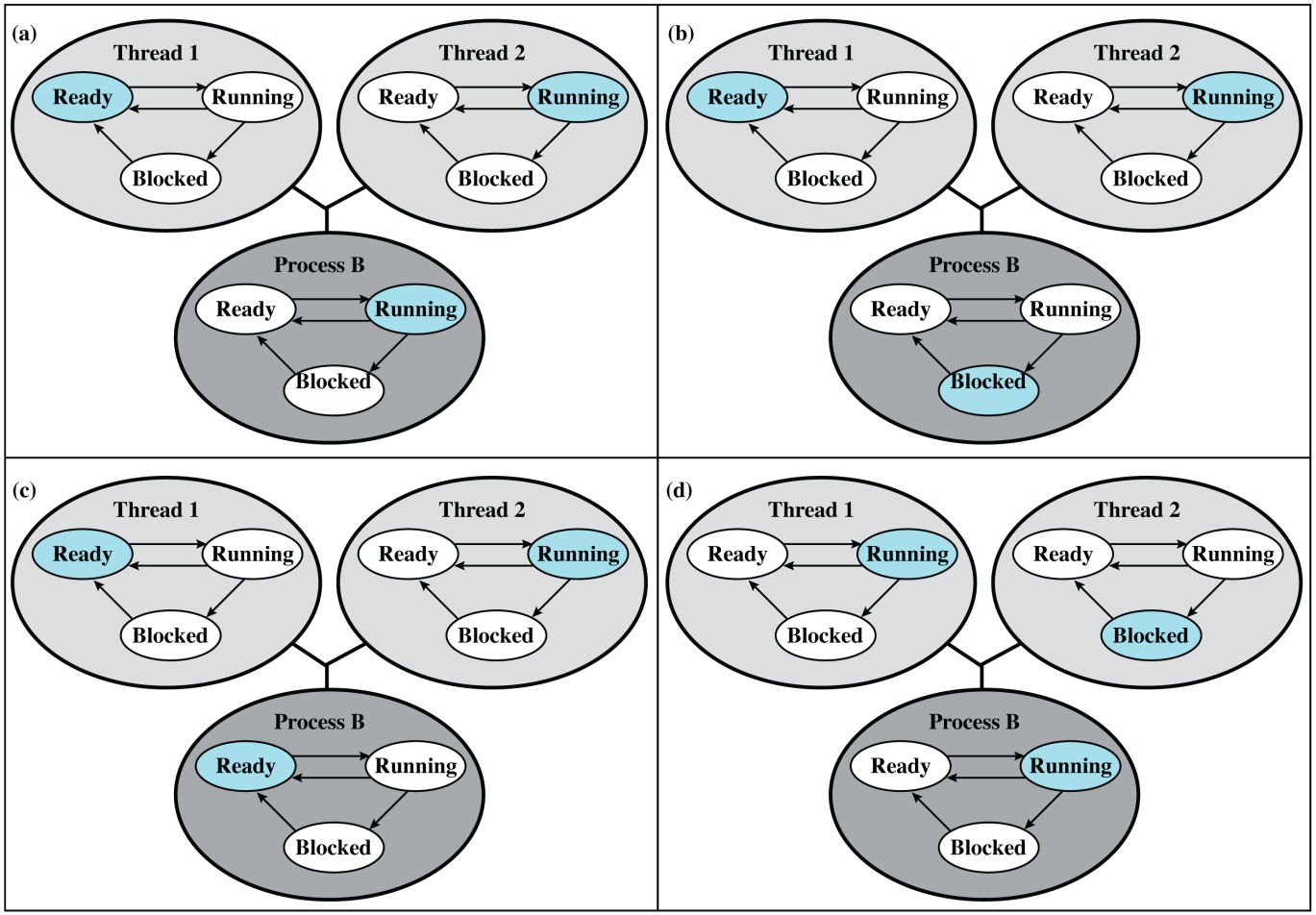
- Benefits of threads
  - Takes less time to create a new thread than a process (generally)
  - Less time to terminate a thread than a process (generally)
  - Switching between two threads takes less time than switching between processes (generally)
  - Threads enhance efficiency in communication between programs
- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is thus maintained in thread-level data structures
  - Suspending a process involves suspending all threads of the process
  - Termination of a process terminates all threads within the process
  - Key states for a thread: Running, Ready, Blocked
  - Thread operations associated to change in state: Spawn, Block, Unblock, Finish

TYPES OF THREADS: USER-LEVEL THREADS

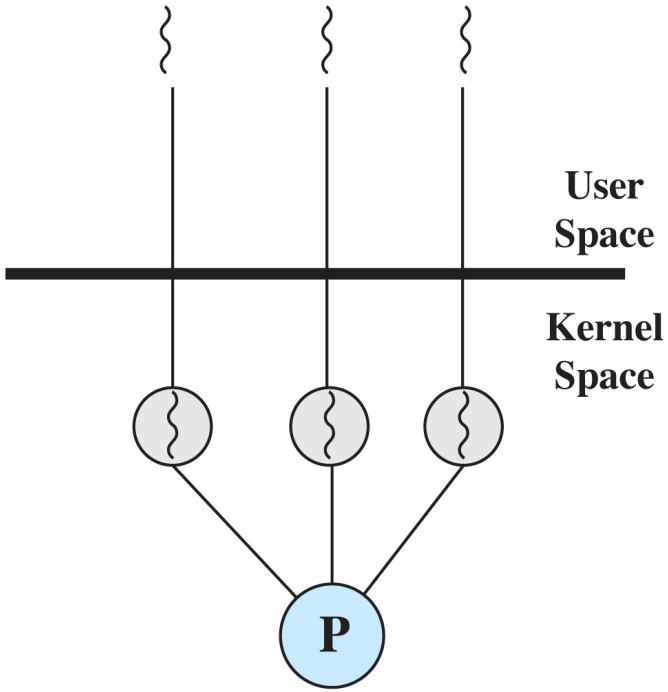
- User-level threads (ULT) - all thread management done by the application
- Kernel is not aware of the existence of threads
- Advantages:
  - Thread switching does not require kernel mode privileges
  - Scheduling can be application specific
  - Can run on any OS
- Disadvantages:
  - Many system calls are blocking $\Rightarrow$ when a ULT execute a system call all the threads are blocked
    (Possible solution: jacketing = converts a blocking system call into a non-blocking system call – library level)
  - A pure ULT application cannot take advantage of multiprocessing
    (Possible solution: write a multi-process application)



ULT AND PROCESS STATES



TYPES OF THREADS: KERNEL-LEVEL THREADS



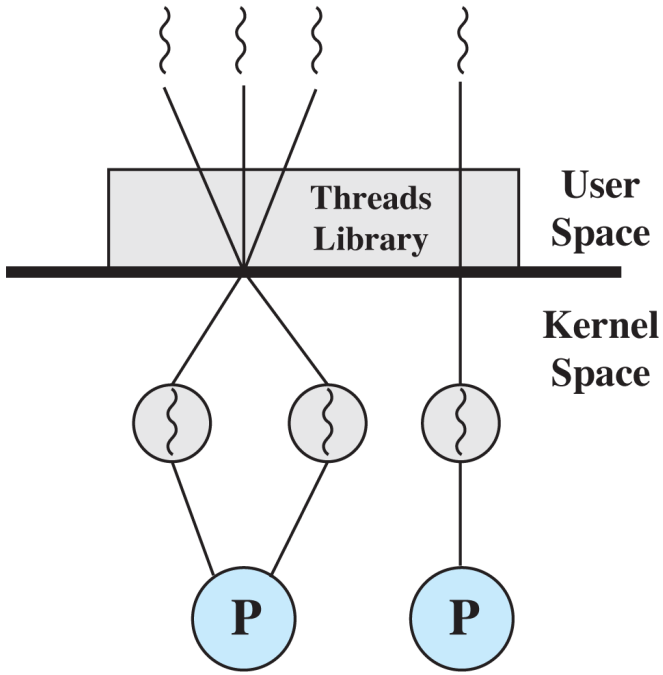
- Kernel-level threads (KLT) - all thread management done by the kernel (application not involved)
- Advantages:
  - The kernel can simultaneously schedule multiple threads from the same process on multiple processors
  - If one thread in a process is blocked, the kernel can schedule another thread of the same process
  - Kernel routines can be multithreaded
- Disadvantages:
  - The transfer of control from one thread to another within the same process requires a mode switch to the kernel (usually more expensive)
- Example: Windows

TYPES OF THREADS: COMBINED APPROACHES

- Thread creation done in user space
- Bulk of scheduling and synchronization of threads done by the application
- Example: Solaris

Relationships between thread and processes:

- 1:1 - each thread of execution is a unique process with its own address space and resources (traditional Unix)
- M:1 - a process defines an address space and dynamic resource ownership; multiple threads may be created and executed within that process (Windows NT, Linux, Mach, etc.)
- 1:M - a thread may migrate from one process environment to another; this allows a thread to be easily moved among distinct systems (clouds, Ra, Emerald)
- M:N - combines attributes of M:1 and 1:M cases (TRIX)

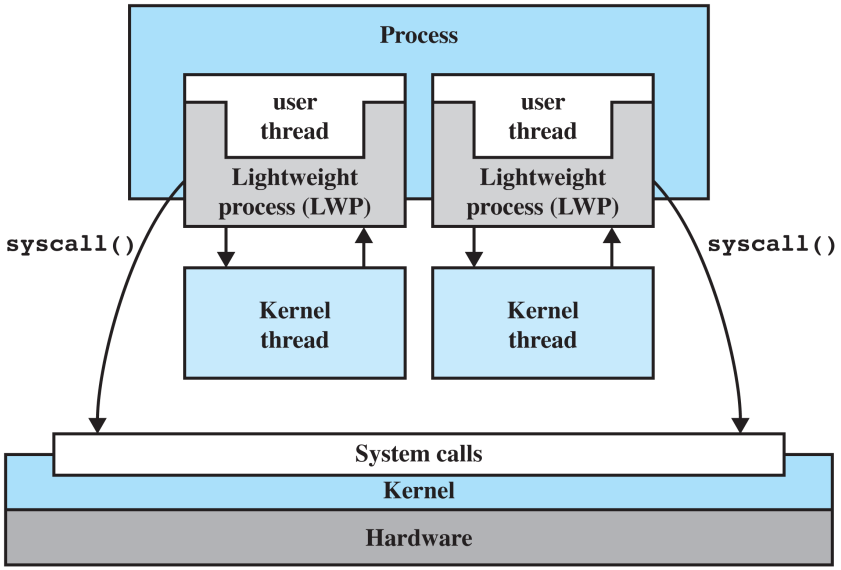


THREAD ISSUES

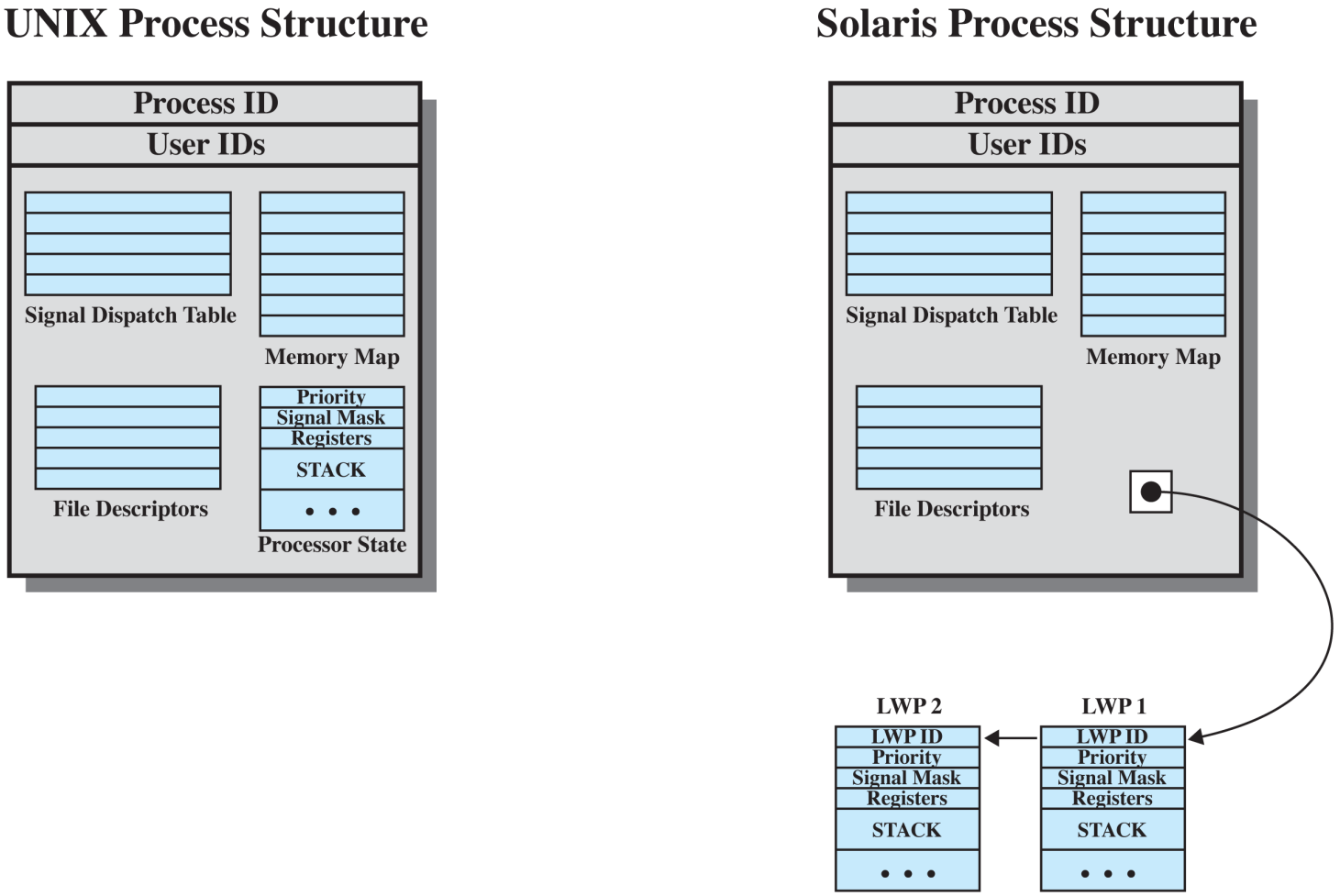
- Semantics of fork() and exec() system calls
  - Does fork() duplicate only the calling thread or all threads?
- Signal handling in multi-threaded processes
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process
- Symmetric multiprocessing issues
  - Threads of any process can run on any processor
  - Sometimes it is advantageous to be more precise
    - E.g., Windows allow to specify soft affinity (the dispatcher tries to assign a ready thread to the same processor it last ran on) or hard affinity (restricts thread execution to certain processors)
    - Soft affinity helps reuse data still in that processor's memory caches from the previous execution of the thread

SOLARIS THREADS

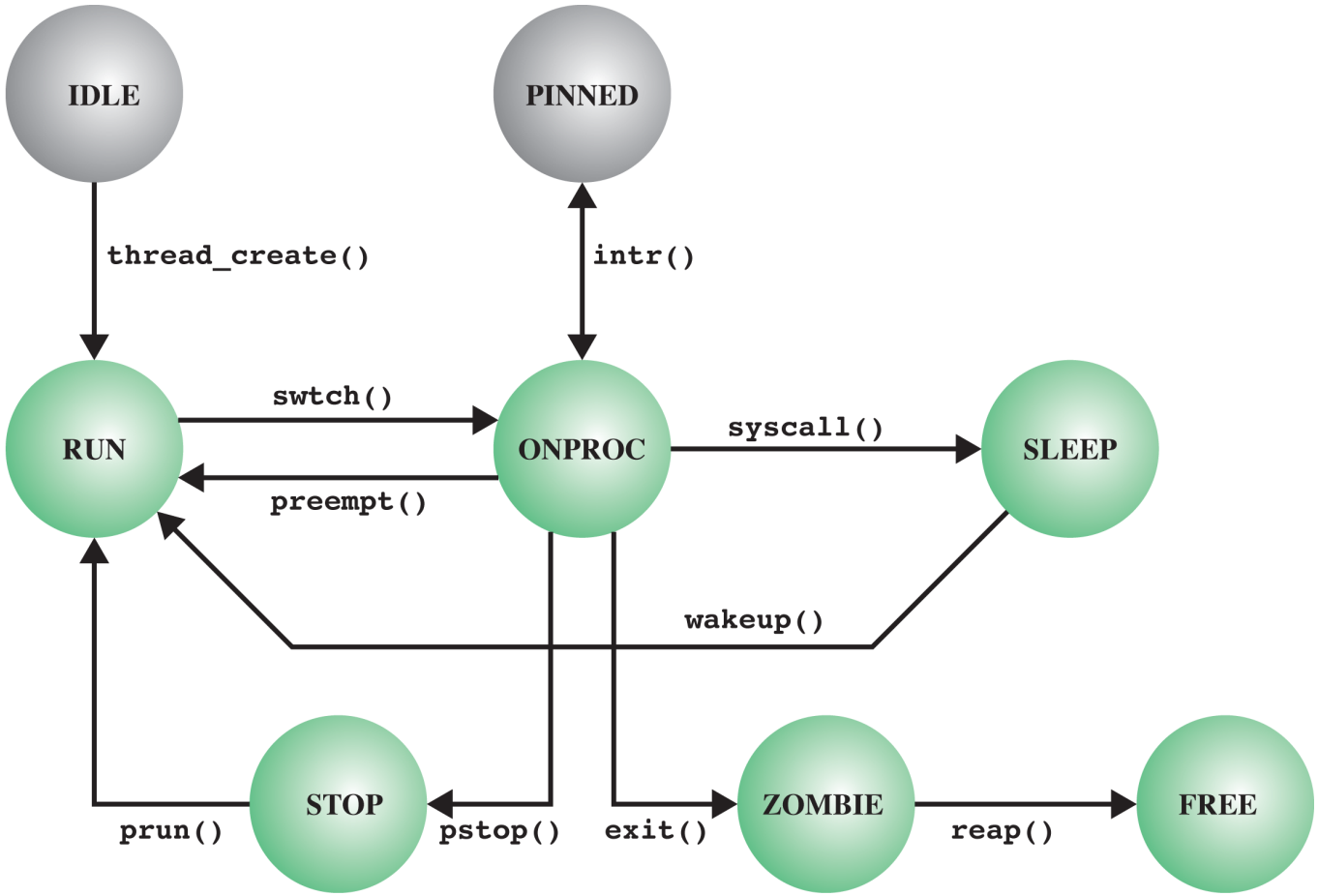
- Process – includes the user's address space, stack, and process control block
- User-level Threads – a user-created unit of execution within a process
- Lightweight Processes (LWP) – a mapping between ULTs and kernel threads
- Kernel Threads – fundamental entities that can be scheduled and dispatched to run on one of the system processors



TRADITIONAL UNIX VERSUS SOLARIS



SOLARIS THREAD STATES

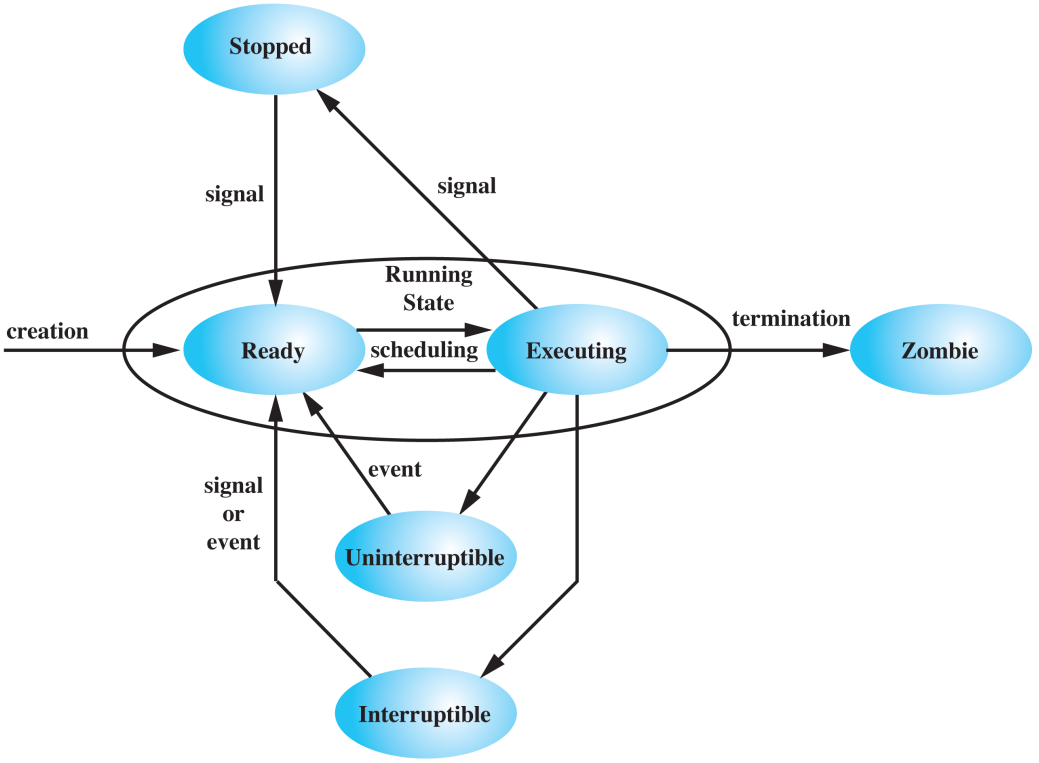


INTERRUPTS AS THREADS

- Most operating systems contain two fundamental forms of concurrent activity:
  - Processes or threads = data manipulation workhorses
  - Interrupts = asynchronous conditions that need attention
- Solaris solution:
  - Set of kernel threads handle interrupts
  - An interrupt thread has its own identifier, priority, context, and stack
  - The kernel controls access to data structures and synchronizes among interrupt threads using mutual exclusion primitives
  - Interrupt threads are assigned higher priorities than all other types of kernel threads

LINUX TASKS

- A process, or task, in Linux is represented by a `task_struct` data structure
- No distinction in kernel between threads and processes
  - ULT mapped into kernel-level processes
  - A new process is created by copying all attributes of the current process
  - The process can also be cloned so that it shares resources
    - The `clone()` system call creates separate stack spaces for each process



LINUX CLONE() FLAGS

CLONE_CLEARID	Clear the task ID
CLONE_DETACHED	The parent does not want a SIGCHLD signal sent on exit
CLONE_FILES	Shares the table that identifies the open files
CLONE_FS	Shares the table that identifies the root directory and the current working directory, as well as the value of the bit mask used to mask the initial file permissions of a new file
CLONE_IDLETASK	Set PID to zero, which refers to an idle task. The idle task is employed when all available tasks are blocked waiting for resources
CLONE_NEWNS	Create a new namespace for the child
CLONE_PARENT	Caller and new task share the same parent process
CLONE_PTRACE	If the parent process is being traced, the child process will also be traced
CLONE_SETTID	Write the TID back to user space
CLONE_SETTLS	Create a new TLS for the child
CLONE_SIGHAND	Shares the table that identifies the signal handlers
CLONE_SYSVSEM	Shares System V SEM_UNDO semantics
CLONE_THREAD	Inserts this process into the same thread group of the parent. If this flag is true, it implicitly enforces CLONE_PARENT
CLONE_VFORK	If set, the parent does not get scheduled for execution until the child invokes the execve() system call
CLONE_VM	Shares the address space (memory descriptor and all page tables)

MAC OS X GRAND CENTRAL DISPATCH

- Provides a pool of available threads
- The developer designates portions of applications, called blocks, that can be dispatched independently and run concurrently
  - A simple extension to any programming language
  - A block defines a self-contained unit of work
  - Enables the programmer to encapsulate complex functions
  - Scheduled and dispatched by queues (first-in-first-out basis)
  - Can be associated with an event source such as a timer, network socket, or file descriptor
- Concurrency is based on the number of cores available as well as the thread capacity of the system

POSIX THREADS

- POSIX standard 1003.1, observed by most other Unix systems
- Characteristics
  - Threads can be created at any time using the system call `pthread_create`
  - Threads execute concurrently, and are preemptible
    - A thread can give up the CPU voluntarily by using the system call `sched_yield` (also available for processes)
  - Each thread has its own copy of local variables, but all threads in a process share global variables and the descriptor table
  - The threads API include functions for coordination and synchronization (including mechanisms to implement critical regions in memory, i.e., without file locks).
  - No mention in the standard whether the threads are user- or kernel-level
- A program that uses threads must include `<pthread.h>` and must be linked with the library `pthread`, i.e.,

```
g++ -lpthread -o foo foo.cc
g++ -lpthread -o tserv tserv.o tcp-utils.o
```

- When working with processes, you need to wory about exclussive access only when accessing files and other such resources
- When using threads, global variables are also shared, so we wory in general about any kind of global resources
- The following mechanisms for coordination and synchronization are available:

  Mutex: Used to provide exclusive access to a shared piece of data.

  – More generally, you can use a mutex to implement a critical region

Operation	System call
Initialization	`pthread_mutex_init`
Enter critical region	`pthread_mutex_lock`
Release critical region	`pthread_mutex_unlock`
Test for availability	`pthread_mutex_trylock`

(Counting) semaphore. Like a mutex, but for n copies of the resource

Instead of:	Use:
`pthread_mutex_init`	`sem_init`
`pthread_mutex_lock`	`sem_wait`
`pthread_mutex_unlock`	`sem_post`
`pthread_mutex_trylock`	`sem_trywait`
	`sem_getvalue`

- Include `<semaphore.h>` to work with semaphores

Condition variable = mutex + condition

- A number of threads need to access a critical region (mutex)
- Once the critical region is acquired, a certain condition has to be met before going any further
- While it waits for the condition, a thread gives up the mutex so that other threads may proceed

CONDITION VARIABLE (EXAMPLE)

- Wait till `x` gets larger than `y`:

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&mut);
while (x <= y) {
  pthread_cond_wait(&cond, &mut); /* mut is released while waiting */
}
/* mut is reacquired */
/* do stuff with x and y */
pthread_mutex_unlock(&mut);
```

- When `x` becomes larger than `y`, the corresponding condition should be signalled:

```
pthread_mutex_lock(&mut);
/* code that changes x and y */
if (x > y) pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mut);
```

CODING EXAMPLES: MUTEX

```
#include <pthread.h>

// lock1, lock2 MUST be global
pthread_mutex_t lock1;
pthread_mutex_t lock2;

pthread_mutex_init(&lock1,NULL);
pthread_mutex_init(&lock2,NULL);

// Do something involving two critical regions, i.e. use
//   pthread_mutex_lock(&lock1)     pthread_mutex_unlock(&lock1)
//   pthread_mutex_lock(&lock2)     pthread_mutex_unlock(&lock2)

// clean up: could call pthread_mutex_destroy
// except that it does nothing
```

CODING EXAMPLES: MUTEX AND THREADS

```
pthread_mutex_t lock1, lock2;

void* do_lock (int n) {
  pthread_mutex_lock(&lock1);
  cout << "Thread " << n << " enters critical.\n";
  sched_yield(); sleep(3);
  pthread_mutex_unlock(&lock1);
  cout << "Thread " << n << " exits critical.\n";
  return NULL;
}
int main () {
  pthread_mutex_init(&lock1,NULL);  pthread_mutex_init(&lock2,NULL);

  pthread_t tt;
  pthread_attr_t ta;
  pthread_attr_init(&ta);
  pthread_attr_setdetachstate(&ta,PTHREAD_CREATE_DETACHED);

  pthread_create(&tt, &ta, (void* (*) (void*))do_lock, (void*)1);
  pthread_create(&tt, &ta, (void* (*) (void*))do_lock, (void*)2);
  pthread_create(&tt, &ta, (void* (*) (void*))do_lock, (void*)3);
  sched_yield(); sleep(60);
}
```

MUTEX AND DEADLOCKS

```
void* do_lock_21 (int n) {
  pthread_mutex_lock(&lock2);
  cout<<"Th. "<<n<<" enters 1.\n";
  sched_yield(); sleep(1);
  pthread_mutex_lock(&lock1);
  cout<<"Th. "<<n<<" enters 2.\n";
  sched_yield(); sleep(3);
  pthread_mutex_unlock(&lock2);
  cout<<"Th. "<<n<<" exits 2.\n";
  pthread_mutex_unlock(&lock1);
  cout<<"Th. "<<n<<" exits 1.\n";
  return NULL;
}
```

```
void* do_lock_12 (int n) {
  pthread_mutex_lock(&lock1);
  cout<<"Th. "<<n<<" enters 1.\n";
  sched_yield(); sleep(1);
  pthread_mutex_lock(&lock2);
  cout<<"Th. "<<n<<" enters 2.\n";
  sched_yield(); sleep(3);
  pthread_mutex_unlock(&lock2);
  cout<<"Th. "<<n<<" exits 2.\n";
  pthread_mutex_unlock(&lock1);
  cout<<"Th. "<<n<<" exits 1.\n";
  return NULL;
}
```

```
int main () {
  [ ... initialize mutexes, thread data ... ]

  pthread_create(&tt, &ta,
             (void* (*) (void*))do_lock_12, (void*)1);
  pthread_create(&tt, &ta,
             (void* (*) (void*))do_lock_21, (void*)2);
  sched_yield(); sleep(60);
}
```

MUTEX AND DEADLOCKS

```
void* do_lock_21 (int n) {
  pthread_mutex_lock(&lock2);
  cout<<"Th. "<<n<<" enters 1.\n";
  sched_yield(); sleep(1);
  pthread_mutex_lock(&lock1);
  cout<<"Th. "<<n<<" enters 2.\n";
  sched_yield(); sleep(3);
  pthread_mutex_unlock(&lock2);
  cout<<"Th. "<<n<<" exits 2.\n";
  pthread_mutex_unlock(&lock1);
  cout<<"Th. "<<n<<" exits 1.\n";
  return NULL;
}
```

```
void* do_lock_12 (int n) {
  pthread_mutex_lock(&lock1);
  cout<<"Th. "<<n<<" enters 1.\n";
  sched_yield(); sleep(1);
  pthread_mutex_lock(&lock2);
  cout<<"Th. "<<n<<" enters 2.\n";
  sched_yield(); sleep(3);
  pthread_mutex_unlock(&lock2);
  cout<<"Th. "<<n<<" exits 2.\n";
  pthread_mutex_unlock(&lock1);
  cout<<"Th. "<<n<<" exits 1.\n";
  return NULL;
}
```

```
int main () {
  [ ... initialize mutexes, thread data ... ]

  pthread_create(&tt, &ta,
              (void* (*) (void*))do_lock_12, (void*)1);
  pthread_create(&tt, &ta,
              (void* (*) (void*))do_lock_21, (void*)2);
  sched_yield(); sleep(60);
}
```

Output:

```
Th. 1 enters 1.
Th. 2 enters 1.
```

. . . nothing happens in the next minute!

TERMINATING A THREAD

- A thread can terminate itself by returning from its main function of by calling `pthread_exit`
- A thread can cancel (i.e., terminate) other threads by sending a cancellation request using `pthread_cancel`
  - Sole argument: the thread being cancelled (`pthread_t`)
  - Depending on its settings, the target thread can ignore the request, honor it immediately, or defer it till it reaches a cancellation point
    * The following POSIX threads functions are cancellation points: `pthread_join`, `pthread_cond_wait`, `pthread_cond_timedwait`, `pthread_testcancel`, `sem_wait`, `sigwait`
    * All other POSIX threads functions are guaranteed not to be cancellation points
    * `pthread_testcancel` does nothing except testing for pending cancellation and executing it
  - When the cancellation is honored the thread being cancelled behaves as if it calls `pthread_exit(PTHREAD_CANCELED)`

CANCELLATION POINTS

- In addition to the cancellation points enumerated above, a number of system calls (basically, all system calls that may block) and library functions that may call these system calls are cancellation points
  - according to the POSIX standard that is
  - however, older implementations do not conform to this
  - workaround:
    * cancellation requests are transmitted to the target thread by sending it a signal
    * the signal will interrupt all blocking system calls, causing them to return immediately with the `EINTR` error
    * so using `pthread_cancel` immediately after a system call is safe and acheives the desired effect
  - it is unclear what is the behaviour of newer implementations (feel free to experiment)

CANCELLATION STATE

- `pthread_setcancelstate` changes the cancellation state for the calling thread
  - that is, whether cancellation requests are ignored or not (possible state values: `PTHREAD_CANCEL_DISABLE`, `PTHREAD_CANCEL_ENABLE`)
  - the old cancellation state is stored and can thus be restored (unless the second argument is 0)
  - prototype: `pthread_setcancelstate(int state, int *oldstate);`
- `pthread_setcanceltype` changes the type of responses to cancellation requests
  - possible behaviour: asynchronous (immediate) or deferred cancellation (`PTHREAD_CANCEL_ASYNCHRONOUS`, `PTHREAD_CANCEL_DEFERRED`)
  - the old cancellation type is stored and can thus be restored (unless the second argument is 0)
  - prototype: `int pthread_setcanceltype(int type, int *oldtype);`
- A thread is created by default with cancellation enabled and deferred

JOINING AND DETACHING

- A thread can wait for the completion of other threads:

```
                                    void* ret;
pthread_create(&tt, ...);   ⇝   pthread_join(tt, &ret);
```

  - `pthread_join` suspends execution of the calling thread until the thread given as argument terminates
  - the return value of the thread (`PTHREAD_CANCELED` if cancelled) is stored in the second argument unless the second argument is 0
  - At most one thread can wait for the termination of a given thread
- A thread can be waited upon only if it is attached
- However, if a thread is attached it does not deallocate any resources unless a `pthread_join` is called on it
  - similar with zombie processes
  - if you do not want to deal with "zombie threads" then you set them to be detached; otherwise you must call `pthread_join` on them