CS 477:

Advanced Operating Systems

Processes & Threads

Administrivia

This week

- Sheet is up, please start signing up
- Lab 0 submission site is up
- Lab 1 will be released soon

Office hours

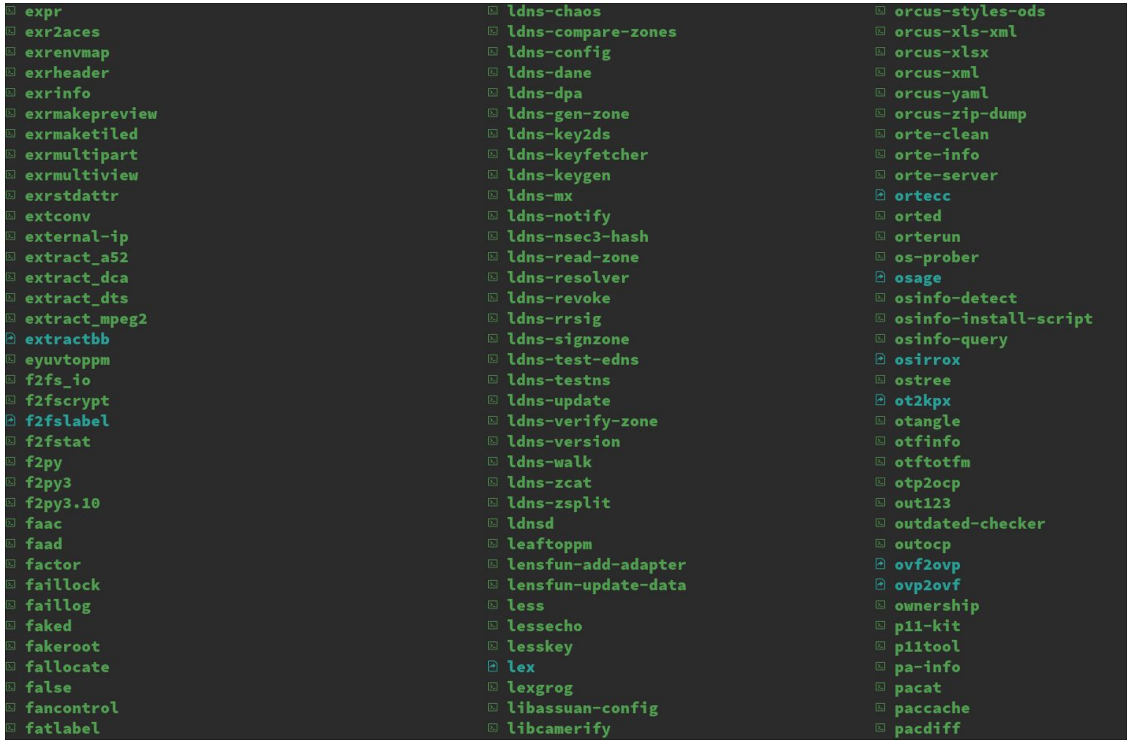
- Each TA will announce their individual office hours from the next week
- You can meet me by appointment

Focus of today's lecture

- Process abstraction
- Thread abstraction
- Process management in Linux
- Another OS abstraction: Lightweight contexts
- Scheduler activations

Programs

- A program consists of code and data
  - Specified in some programming language
- Typically stored in the file on the disk
- A program is (can be) an executable file



“Executable” file

- An executable file contains:
  - Executable code: CPU instructions
  - Data: Information manipulated by these instructions
- Obtained by compiling a program

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  char *input = argv[1]; // Reading from standard input.
  while (1) {
    printf("%s\n", input); // Printing the input infinitely
  }
  return 0;
}
```

From program to processes

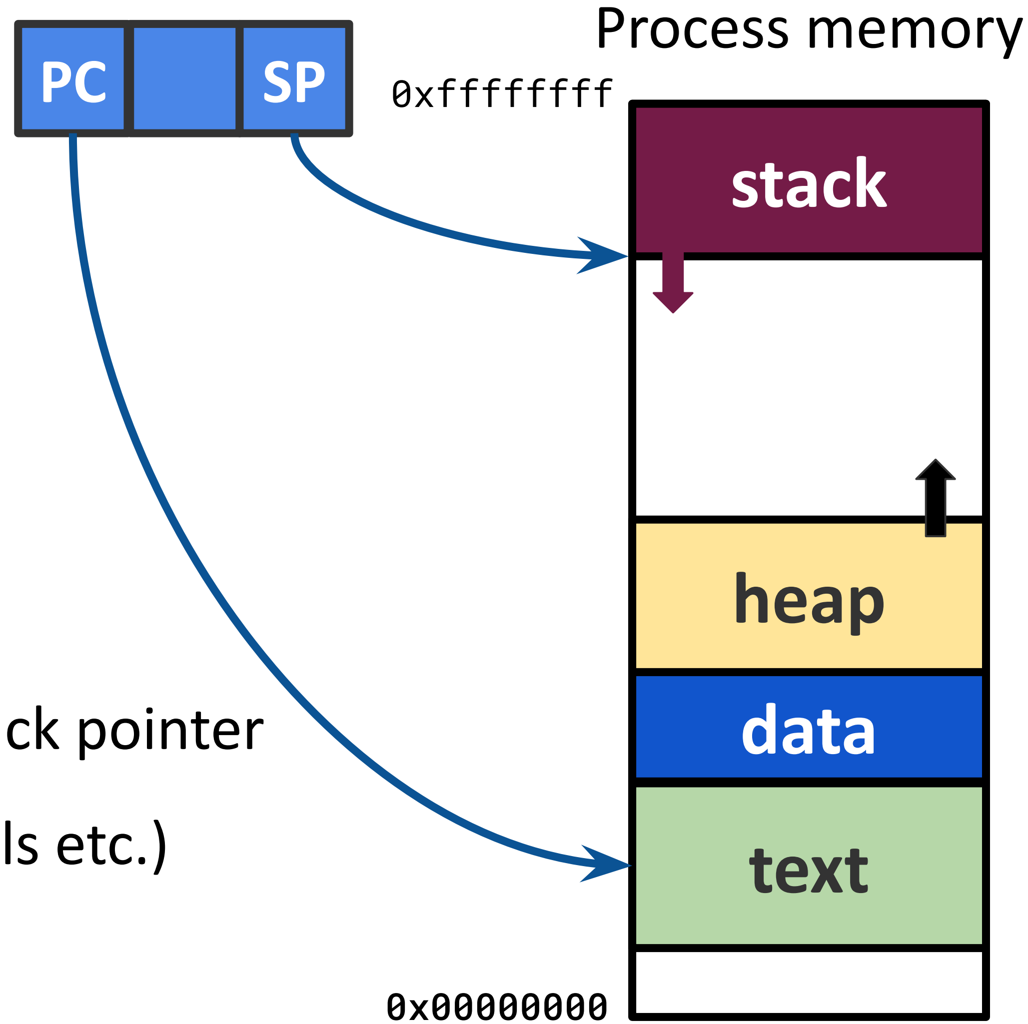
- “Running a program” → creating a process
  - When we run an executable, the OS creates a process
- A process is an **instance** of an executable

What is a process

- **A basic unit of protection**
- Java analogy:
  - Class → "program" (static)
  - Object → "process" (dynamic)
- Every process has a unique ID (PID)

What constitutes a process?

- A unique identifier: Process ID (PID)
- Memory image:
  - Code and data (static)
  - Stack and heap (dynamic)
- CPU context: registers
  - Program counter, current operands, stack pointer
- Kernel resources (open files, pending signals etc.)
- Threads



PC: Program counter; SP: Stack pointer

Focus of today's lecture

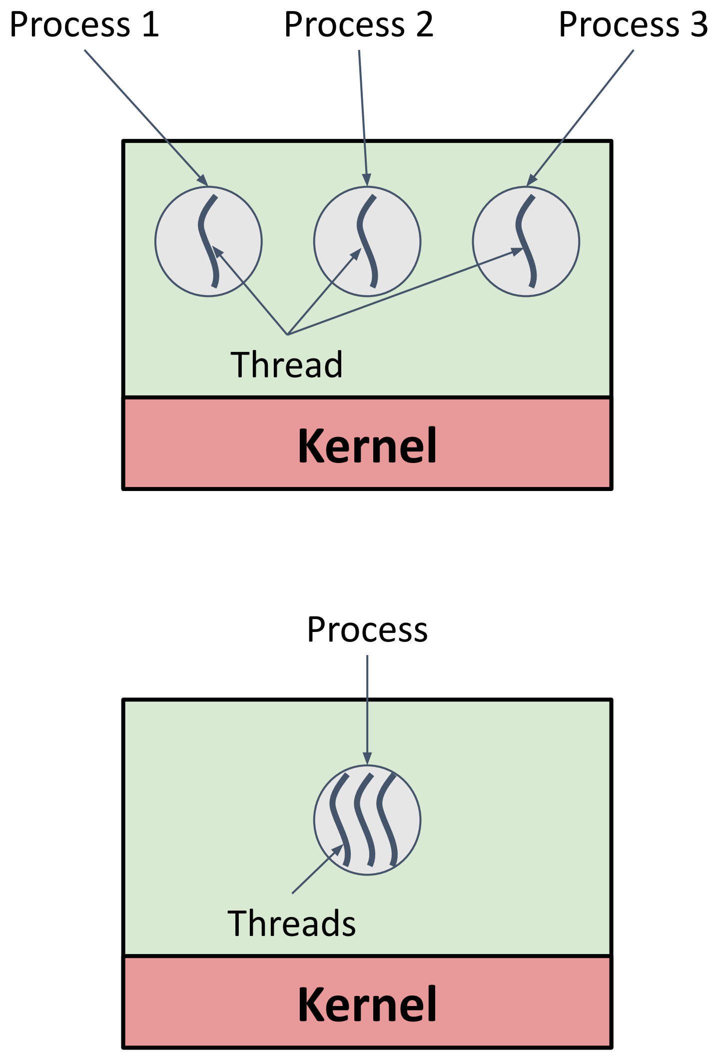
- Process abstraction
- **Thread abstraction**
- Process management in Linux
- Another OS abstraction: Lightweight contexts
- Scheduler activations

Fork: An abstraction for protection

Threads: An execution context

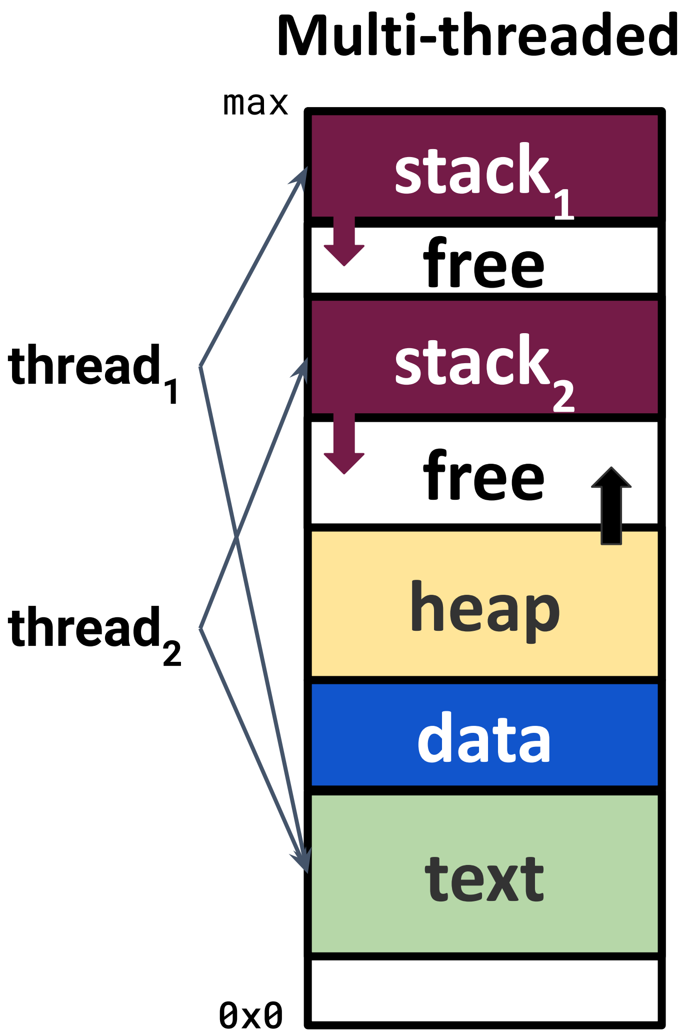
Threads

- An execution context: run sequence of instructions
- A thread has its own:
  - Thread ID (TID)
  - Set of registers including CP & SP
  - Stack
- Threads share the address space
  - Text, data, heap
- Separates the virtual concept of process from its execution state



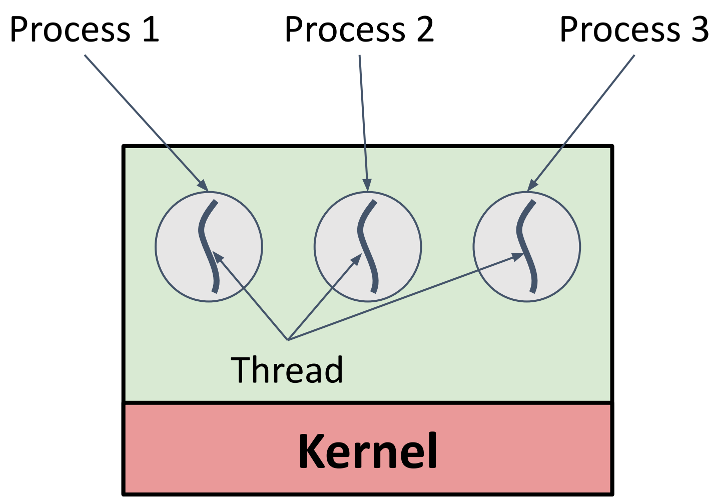
Why do we need threads?

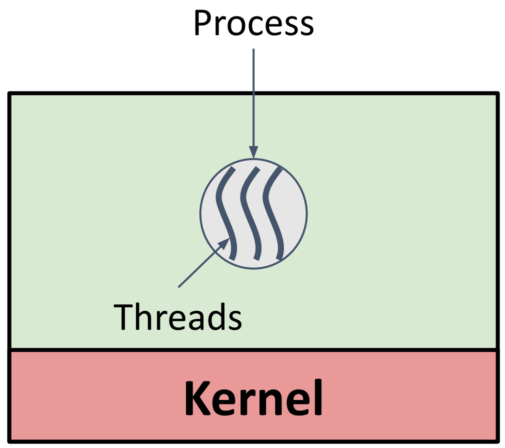
- Threads express the opportunity of concurrency and parallelism
- Improves program structure
  - Divide large tasks across several cooperative threads
- Throughput
  - By overlapping computation with IO operations
- Responsiveness
  - Can handle concurrent events
- Resource sharing
- Utilization of multi-core architectures
  - Allows building parallel programs



Processes vs threads

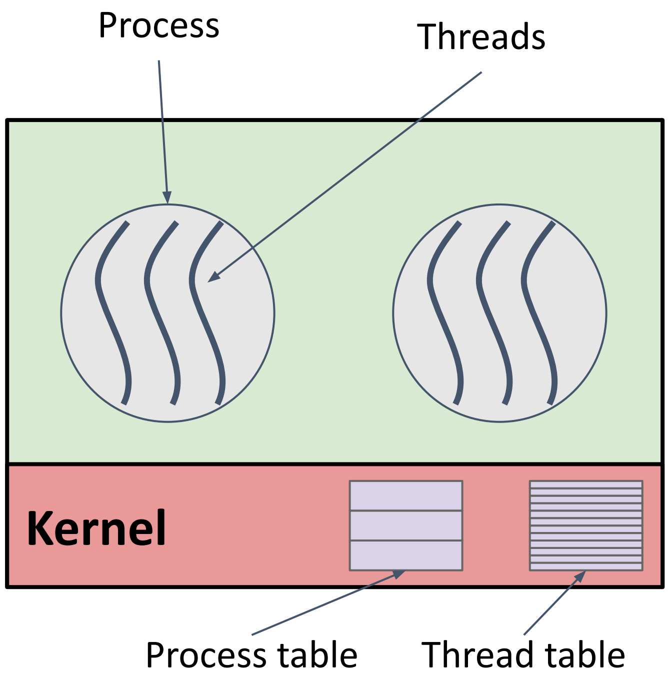
- A thread is bound to a single process
- A process can have multiple threads
- Sharing data between threads is cheap
  - All threads share the same address space
- Threads are the unit of scheduling
- Processes are containers in which threads execute
  - PID, address space, user and group ID, open file descriptors, current working directory, etc.





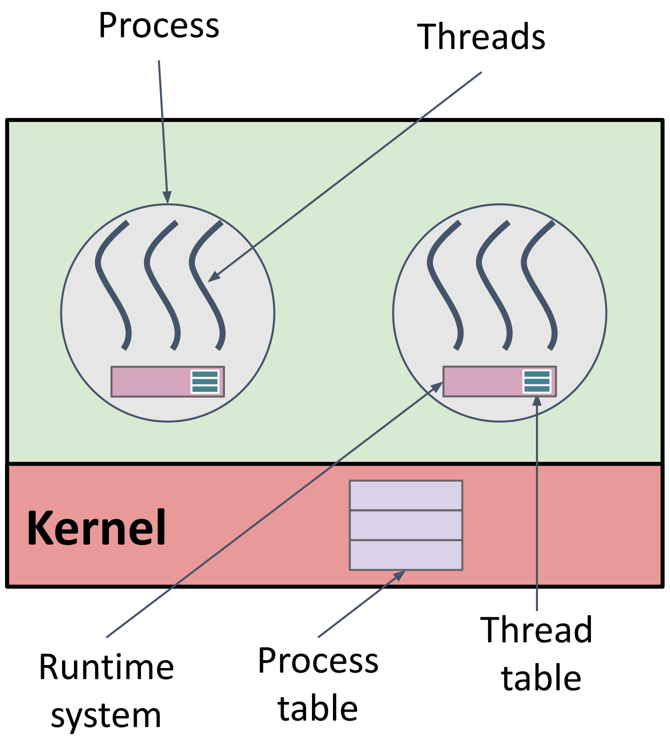
Kernel-level threads: OS managed

- OS manages threads and processes
- All thread operations are implemented in the kernel
- Thread creation/management requires system calls
- OS schedules all threads
- Creating threads is cheaper than creating processes
- Windows, Linux, Solaris, Mac OS, AIX, HP-AUX



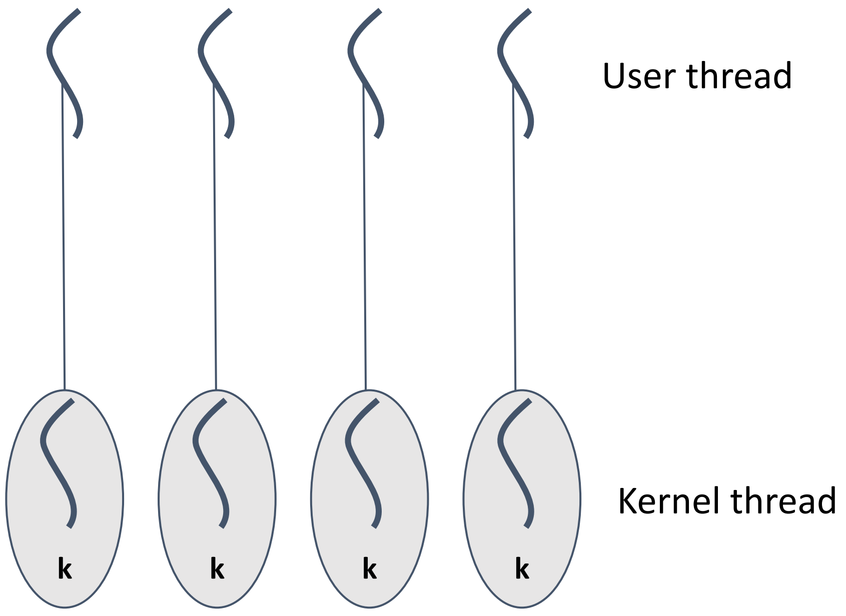
User-level threads: Runtime/application managed

- A library linked into the program, which manages threads
- Threads are invisible to the OS
- All the thread operations are done via procedure calls
  - (no kernel involvement)
- Small and fast:
  - 10–100x faster than kernel-level threads
- Portable
- Tunable to meet application needs
- Java, go, erlang, Node.js, fibers in C++



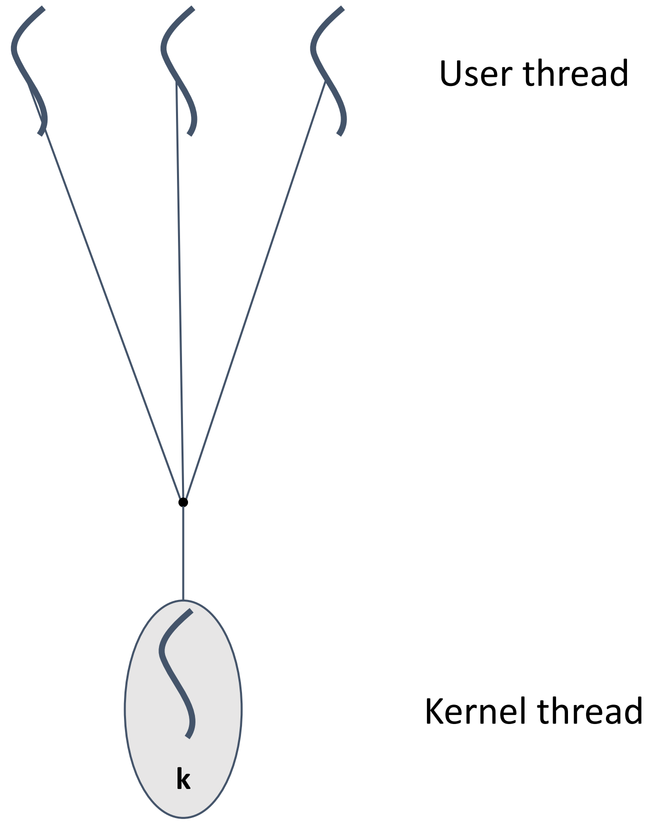
Threading model: 1:1

- Each user-level thread maps to a kernel thread
- Most popular design
- Windows XP/7/10, OS/2, Linux, Solaris v9+



Threading model: N:1

- Many user-level threads map to a single kernel thread
- Used on systems that do not support kernel-level threads
- Solaris green threads, GNU portable threads

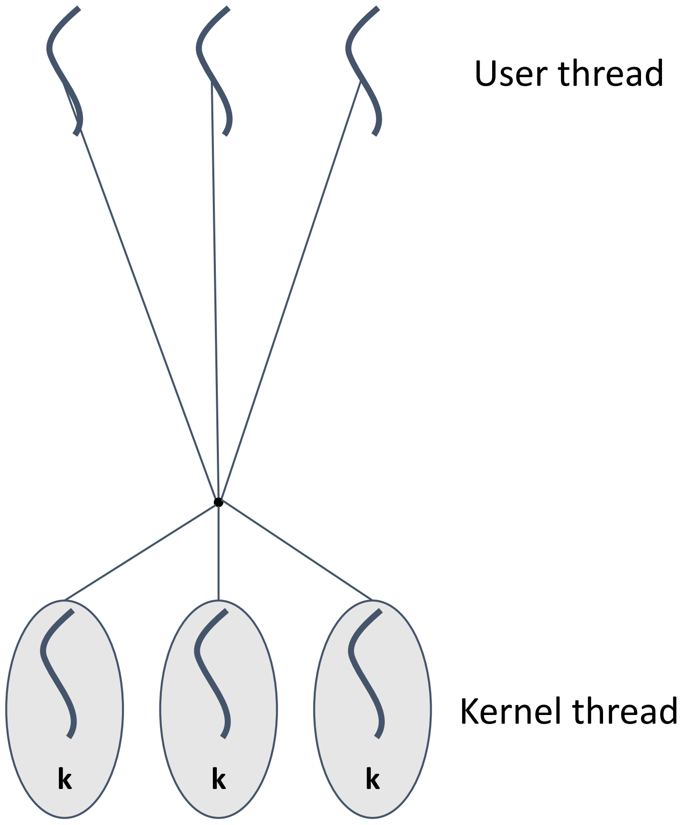


Threading model: N:1

- Allows many user-level threads to be mapped to several kernel-level threads

- Allows the OS to create a sufficient number of kernel threads

- Solaris (before v9), IRIX, HP-UX, Tru64



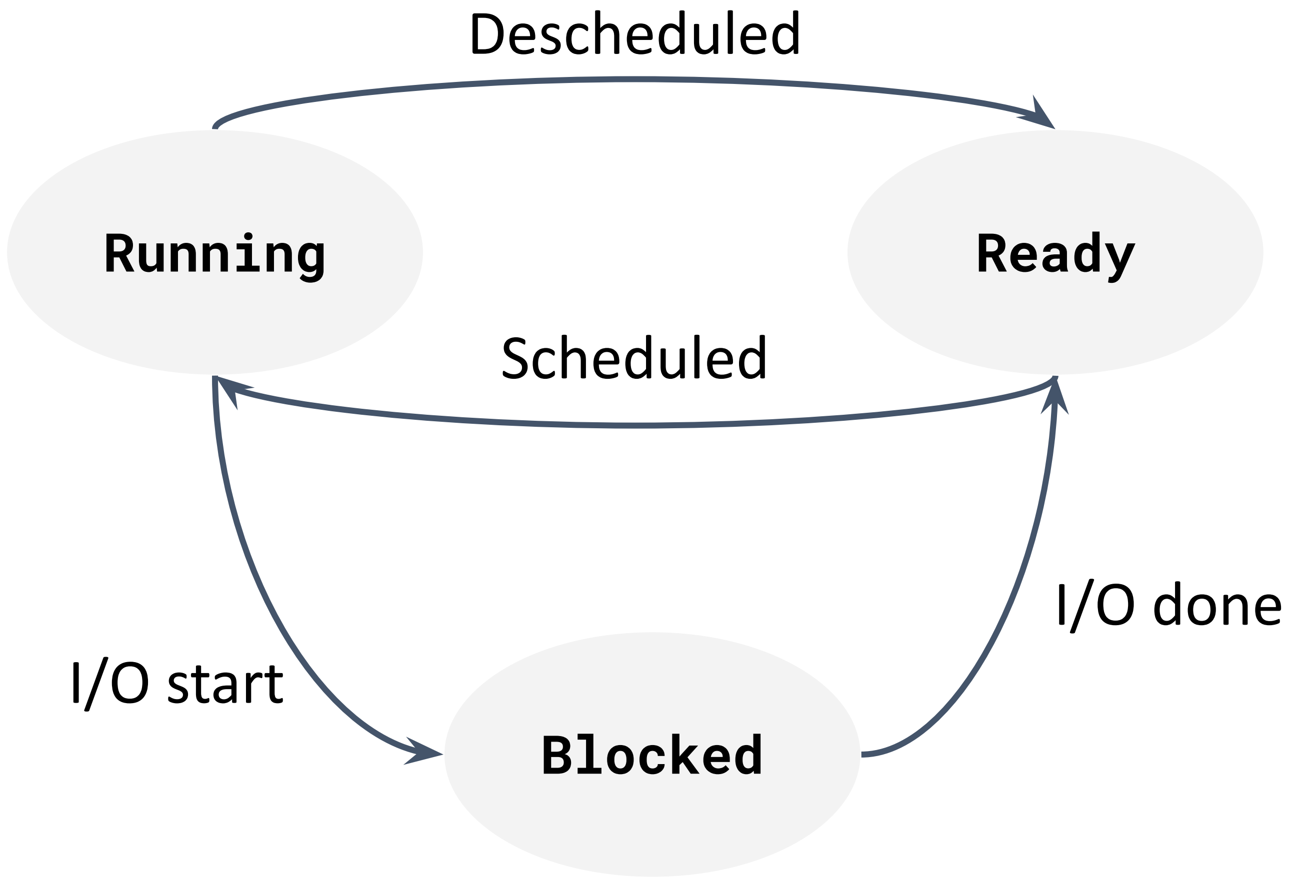
Focus of today's lecture

- Process abstraction
- Thread abstraction
- **Process management in Linux**
- Notifying applications with explicitly calls (Scheduler activations)
- Another OS abstraction: Lightweight contexts

Process state transition

A process can be in one of several states during its life cycle:

- **Running**
- **Ready**
- **Blocked**

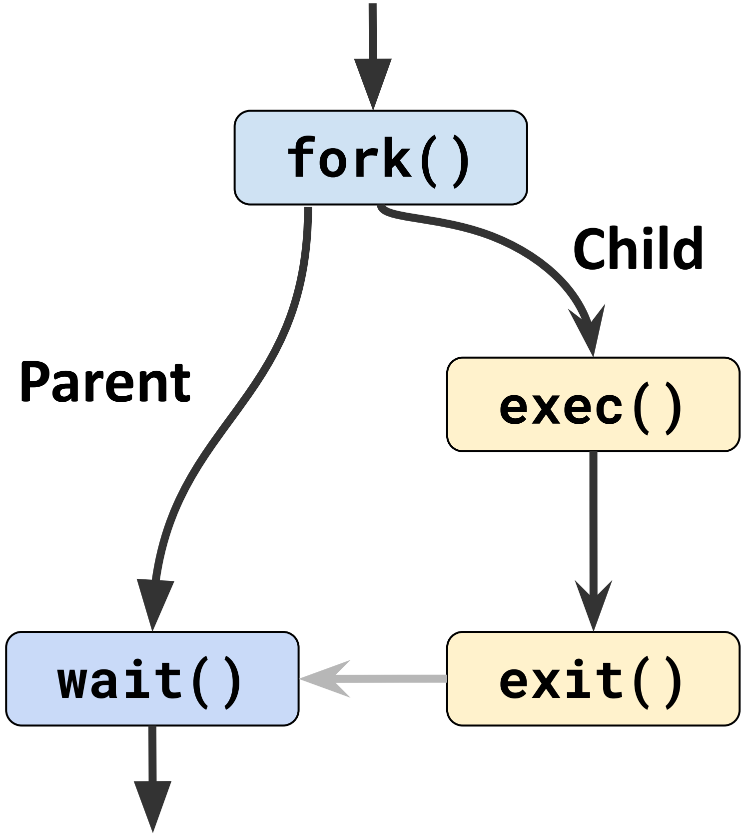


Process from applications' view

A parent process executes

The parent process uses **`fork()`** to create a new **child** process

A child process is created, which is a **copy of its parent**



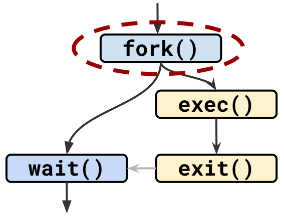
A child process may use **`exec()`** API to replace the parent process' memory space with a new program

The parent process <u>may use</u> **`wait()`** to wait for the **child** process to terminate

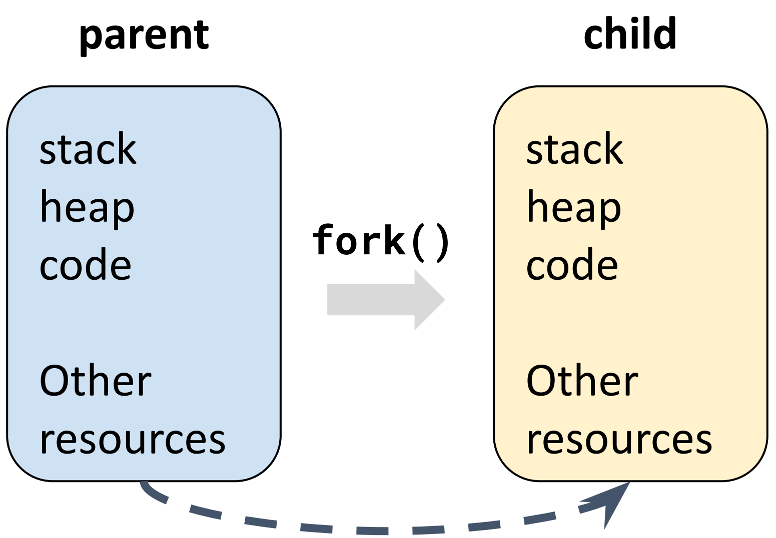
The child process uses the **`exit()`** API to terminate itself

The **parent process** continues to execute

`fork()`: Creating a copy of itself

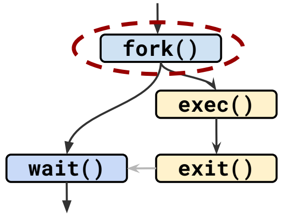


- OS allocates data structures for the child process
- OS *makes a copy* of the caller's address space
- OS also copies other states, such as file descriptors, from the parent process
- The state of the child process is set to READY
- Parent and child execute in their **own separate copy** of the memory (address space)
- `fork` is implemented by the OS



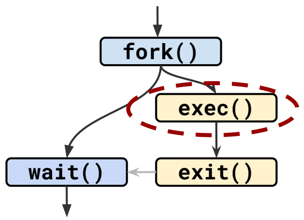
The child process is a copy of the parent process

`fork()` uses copy-on-write (COW)



- Parent’s memory can be huge
- Kernel adopts a **lazy** approach to copying memory: COW
  - Kernel duplicates the parent’s page tables
  - Changes the page table access bits to *read-only*
  - When a page is accessed for *write operations*, that page is copied and the page table entry is changed to *read-write*
- `fork()` becomes fast by delaying or altogether preventing copying of data
- `fork()` saves memory by sharing read-only pages among descendants

exec(): Executing a new program

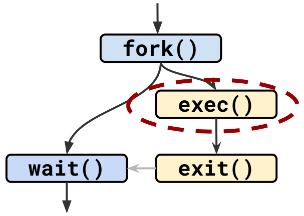


- After **`fork()`**, <u>both parent and child</u> execute the **same program**
  - Suppose we want to run another program
  - If only used **`fork()`**, we need to combine code for parent and child together
    - Becomes difficult, as combining multiple code paths is not easy
    - Ex: Shell program with your own program or existing ones (check **`/bin`**)

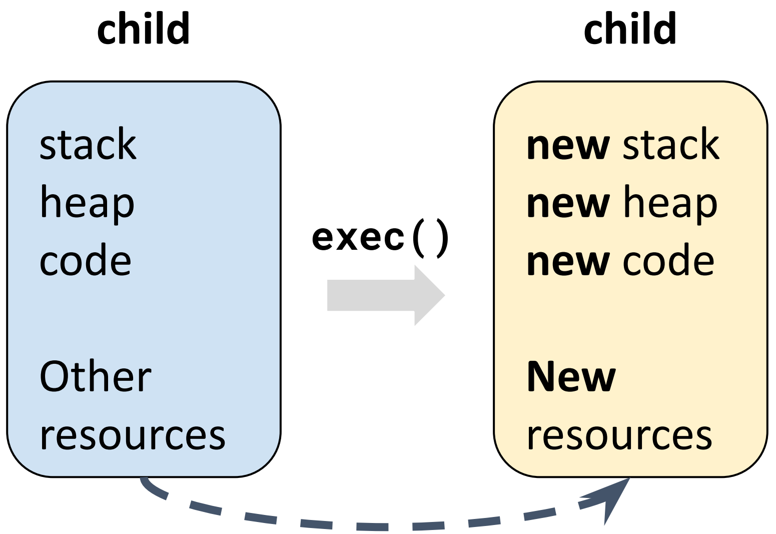
`exec()` loads a new program in the context of an already running process (child),

replacing the previous **executable program**

exec( ): Executing a new program (contd ...)

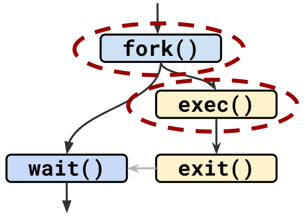


- **exec( )** replaces the memory (address space), loads new program from the disk
  - Code, data, and heap and stack are replaced by the new program
- No new PID allocation
- STDIN, STDOUT, STDERR are kept that allows parent to redirect/rewire child's output
- The new program can pass command line arguments and environment



After exec, new image replaces the old image, except PID, STDIN, STDOUT, and STDERR

Why do we need `fork()` and `exec()`?

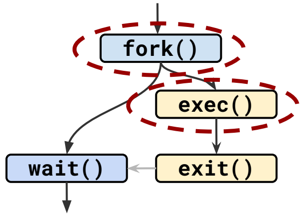


Assume a user wants to start a different program. For that, the OS needs to create a new process and create a new address space to load the program

Let’s divide and conquer:

- `fork()` creates a new process (replica) with a copy of its own address space
- `exec()` replaces the old program image with a new program image

Why do we need `fork()` and `exec()`?



Multiple programs can run simultaneously

  Better utilization of hardware resources

Users can perform various operations between fork() and exec() calls to enable various use cases:

- To redirect standard input/output:
  - `fork`, `close/open file descriptors`, `exec`

  open/close are special file-system calls

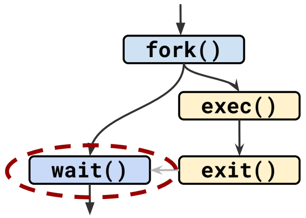
- To switch users:
  - `fork`, `setuid`, `exec`

  Set user ID (change user who can be the owner of the process)

- To start a process with a different current directory:
  - `fork`, `chdir`, `exec`

  Go to a specified directory

`wait()`: Waiting for a child process



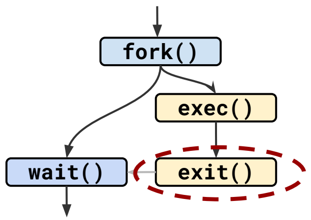
- Child processes are **tied** to their parent
- There exists a hierarchy among processes on forking

A parent process uses **`wait()`** to **suspend its execution** until **one of its children terminates**. The parent process then gets the exit status of the terminated child

```
pid_t wait (int *status);
```

- If **no child** is running, then the wait() call has **no effect** at all
- Else, `wait()` **suspends the caller until one of its children terminates**
- Returns the PID of the terminated child process

`exit()`: Terminating a process

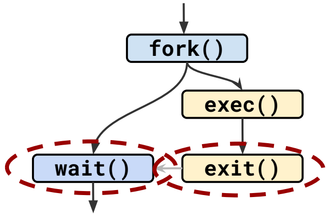


When a process terminates, it executes **`exit()`**, either directly on its own, or indirectly via library code

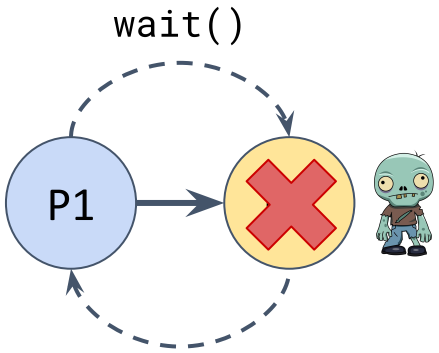
```
void exit (int status);
```

- The call has no return value, as the process terminates after calling the function
- The `exit()` call resumes the execution of a **waiting parent** process

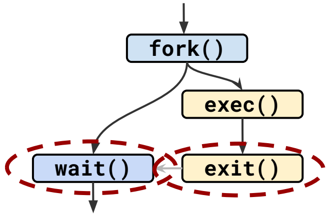
Waiting for children to die …



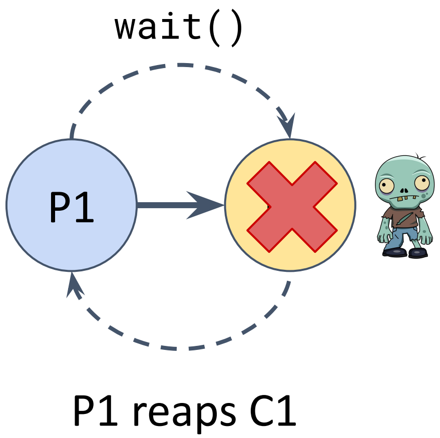
- Scenarios under which a process terminates
  - By calling `exit()` itself
  - OS terminates a misbehaving process
- Terminated process exists as a zombie
- When a parent process calls `wait()`, the zombie child is cleaned up or "**reaped**"
- If a parent terminates before child, the child becomes an **orphan**
  - **`init`** (pid: 1) process adopts orphans and reaps them



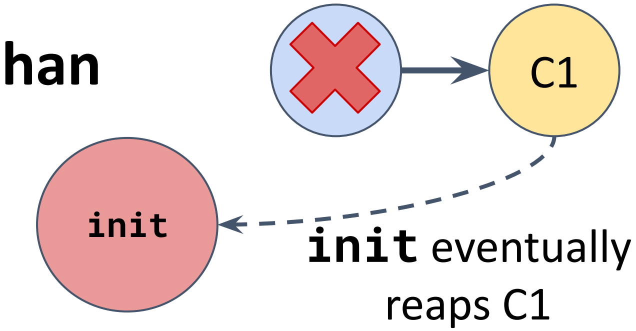
Waiting for children to die …



- Scenarios under which a process terminates
  - By calling `exit()` itself
  - OS terminates a misbehaving process
- Terminated process exists as a zombie
- When a parent process calls `wait()`, the zombie child is cleaned up or "**reaped**"



- If a parent terminates before child, the child becomes an **orphan**
  - **`init`** (pid: 1) process adopts orphans and reaps them



A tree of processes

- Each process has a parent process
    - **init** is the first process (pid: 1) without any parent process
- A process can have many child processes
- Each process again can have child processes

```
    PID▿USER      PRI  NI  VIRT   RES   SHR S  CPU% MEM%   TIME+  Command (merged)
      1 root       20   0  173M 11760  6472 S   0.0  0.0  0:30.33 systemd splash --system --deserialize=17
   2062 root       20   0  457M 10628  7392 S   0.0  0.0  0:00.72 ├─ udisksd
   2095 root       20   0  457M 10628  7392 S   0.0  0.0  0:00.00 │  ├─ cleanup│udisksd
   2078 root       20   0  457M 10628  7392 S   0.0  0.0  0:00.00 │  ├─ probing-thread│udisksd
   2070 root       20   0  457M 10628  7392 S   0.0  0.0  0:00.05 │  ├─ gdbus│udisksd
   2068 root       20   0  457M 10628  7392 S   0.0  0.0  0:00.09 │  ├─ pool-spawner│udisksd
   2067 root       20   0  457M 10628  7392 S   0.0  0.0  0:00.00 │  └─ gmain│udisksd
   1771 sanidhya   20   0 20068  8376  5188 S   0.0  0.0  0:12.33 ├─ systemd --user
2948990 sanidhya   20   0 32.7G  2612  2328 S   0.0  0.0  0:00.04 │  ├─ chrome_crashpad_handler --monitor-self-annotation=ptype=
2948992 sanidhya   20   0 32.7G  2612  2328 S   0.0  0.0  0:00.01 │  │  ├─ chrome_crashpad_handler --monitor-self-annotation=pty
2948991 sanidhya   20   0 32.7G  2612  2328 S   0.0  0.0  0:00.02 │  │  └─ chrome_crashpad_handler --monitor-self-annotation=pty
2412161 sanidhya   20   0 2639M  339M 22368 S   0.0  1.1  1:16.16 │  ├─ xdg-desktop-portal-gnome
2412242 sanidhya   20   0 2639M  339M 22368 S   0.0  1.1  0:00.00 │  │  ├─ xdg-desktop-portal-gnome
2412241 sanidhya   39  19 2639M  339M 22368 S   0.0  1.1  0:00.00 │  │  ├─ xdg-desktop-portal-gnome
2412191 sanidhya   20   0 2639M  339M 22368 S   0.0  1.1  0:00.00 │  │  ├─ xdg-desktop-portal-gnome
2412188 sanidhya   20   0 2639M  339M 22368 S   0.0  1.1  0:04.34 │  │  ├─ xdg-desktop-portal-gnome
2412187 sanidhya   20   0 2639M  339M 22368 S   0.0  1.1  0:07.64 │  │  ├─ xdg-desktop-portal-gnome
2412186 sanidhya   20   0 2639M  339M 22368 S   0.0  1.1  0:00.47 │  │  ├─ xdg-desktop-portal-gnome
2412185 sanidhya   20   0 2639M  339M 22368 S   0.0  1.1  0:00.00 │  │  ├─ xdg-desktop-portal-gnome
2412184 sanidhya   39  19 2639M  339M 22368 S   0.0  1.1  0:00.00 │  │  ├─ xdg-desktop-portal-gnome
2412183 sanidhya   20   0 2639M  339M 22368 S   0.0  1.1  0:00.00 │  │  ├─ xdg-desktop-portal-gnome
2412182 sanidhya   20   0 2639M  339M 22368 S   0.0  1.1  0:00.00 │  │  ├─ xdg-desktop-portal-gnome
2412181 sanidhya   20   0 2639M  339M 22368 S   0.0  1.1  0:00.00 │  │  ├─ xdg-desktop-portal-gnome
```

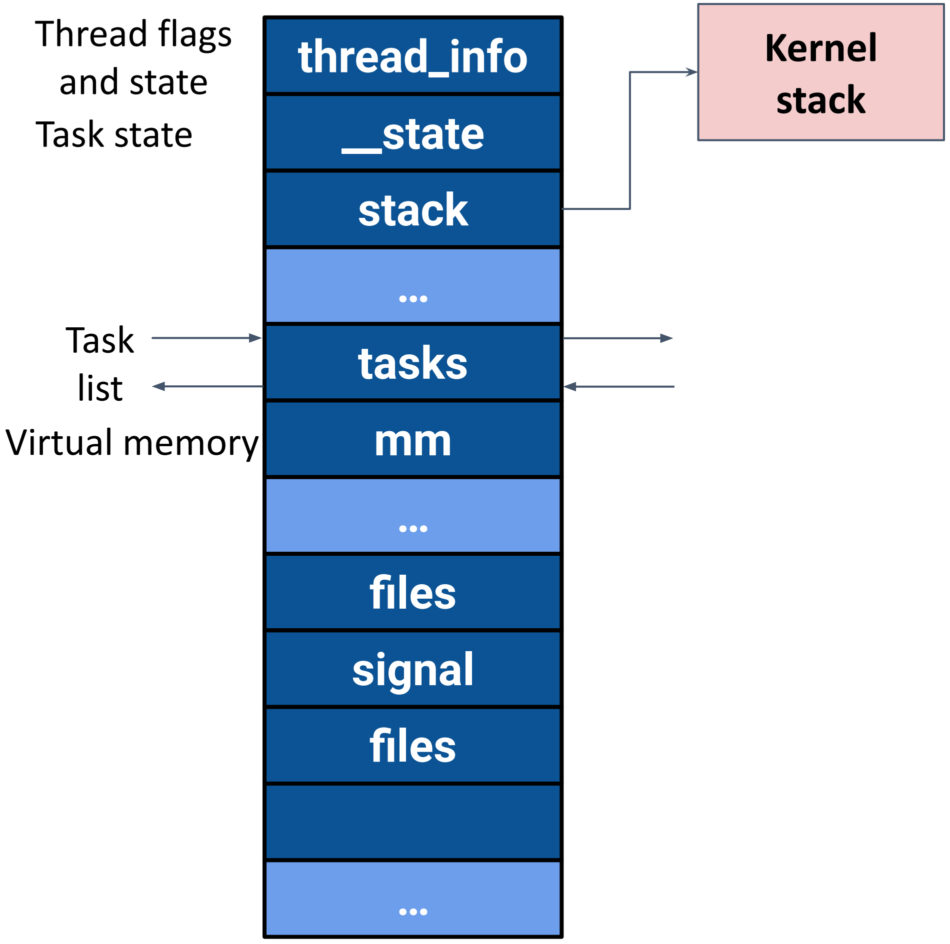
Focus of today's lecture

- Process abstraction
- Thread abstraction
- **Process management in Linux**
- Another OS abstraction: Lightweight contexts
- Scheduler activations

Linux process descriptor: task_struct

- Tasks represent both processes and threads
- Each task is described using a task_struct structure
- It is more than 3.5 KB in size

```c
/* linux/include/linux/sched.h */
struct task_struct {
    struct thread_info   thread_info;  /* thread information */
    volatile long        __state;      /* task status: TASK_RUNNING, etc */
    void                 *stack;       /* stack of this task */
    int                  prio;         /* task priority */
    struct sched_entity  se;           /* information for processor scheduler */
    cpumask_t            cpus_mask;    /* bitmask of CPUs allowed to execute */
    struct list_head     tasks;        /* a global task list */
    struct mm_struct     *mm;          /* memory mapping of this task */
    struct task_struct   *parent;      /* parent task */
    struct list_head     children;     /* a list of child tasks */
    struct list_head     sibling;      /* siblings of the same parent */
    struct files_struct  *files;       /* open file information */
    struct signal_struct *signal;      /* signal handlers */
    /* ... */
    /* NOTE: In Linux kernel, process and task are interchangably used. */
}; /* TODO: Let's check `pstree` output. */
```



Threads in Linux (user space)

- Linux implements all threads as standard processes
  - A thread is just another process sharing some information with other processes
    - Each thread has its own task_struct
- **clone()** system call for creating processes and threads
  - For threads: CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND for cloning the information from the parent process

Threads in Linux (user space)

- Linux implements all threads as standard processes
  - A thread is just another process sharing some information with other processes
    - Each thread has its own task_struct
- Uses 1:1 threading model
- **clone()** system call for creating processes and threads
  - For threads: CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND for cloning the information from the parent process

Threads in Linux (user space)

- One can define a type of thread one wants to create:
  - Setting the flags argument in the clone() system call
  - CLONE_VM: parent and child share address space
  - CLONE_FILES: parent and child share open files
  - CLONE_FS: parent and child share file system information
  - CLONE_SIGHAND …

check clone system call: **man 2 clone**

Thread group in Linux (user space)

- A set of threads that act as a whole with regards to some system calls
- The first thread (task) in a process becomes the thread group leader
  - A new thread created with CLONE_THREAD is placed in the same thread group as the calling thread
- Handing process-based system calls:
  - getpid() returns the PIDI of the thread group leader (t->tgid)
  - On exec(), all threads besides thread group leader are terminated, and the new program is executed in the thread group leader
  - After all of the threads in a thread group terminate, a SIGCHLD signal is sent to the parent process
  - Signals may be sent to a thread group as a whole

Kernel Threads

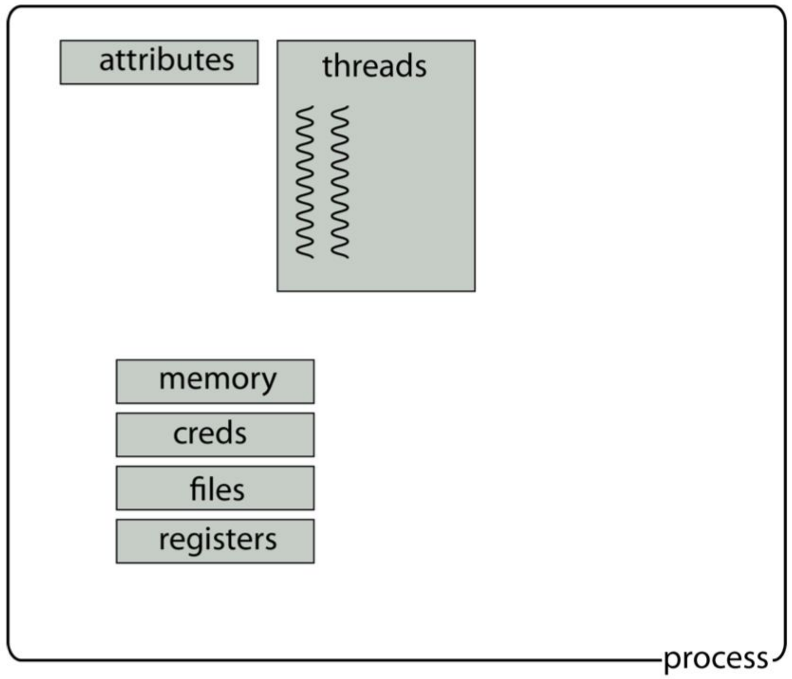
- Runs entirely in the kernel space and performs background operations
- Very similar to user space threads
  - Schedulable entities like regular processes, but never switch to user space
- They do not have their own address space (mm in task_struct is NULL)
- Kernel threads are all forked from the kthreadd thread (PID 2)
- Use cases (ps --ppid 2)
  - Work queue (*kworker*)
  - Load balancing among CPU (*migration*)

Focus of today's lecture

- Process abstraction
- Thread abstraction
- Process management in Linux
- **Another OS abstraction: Lightweight contexts**
- Scheduler activations

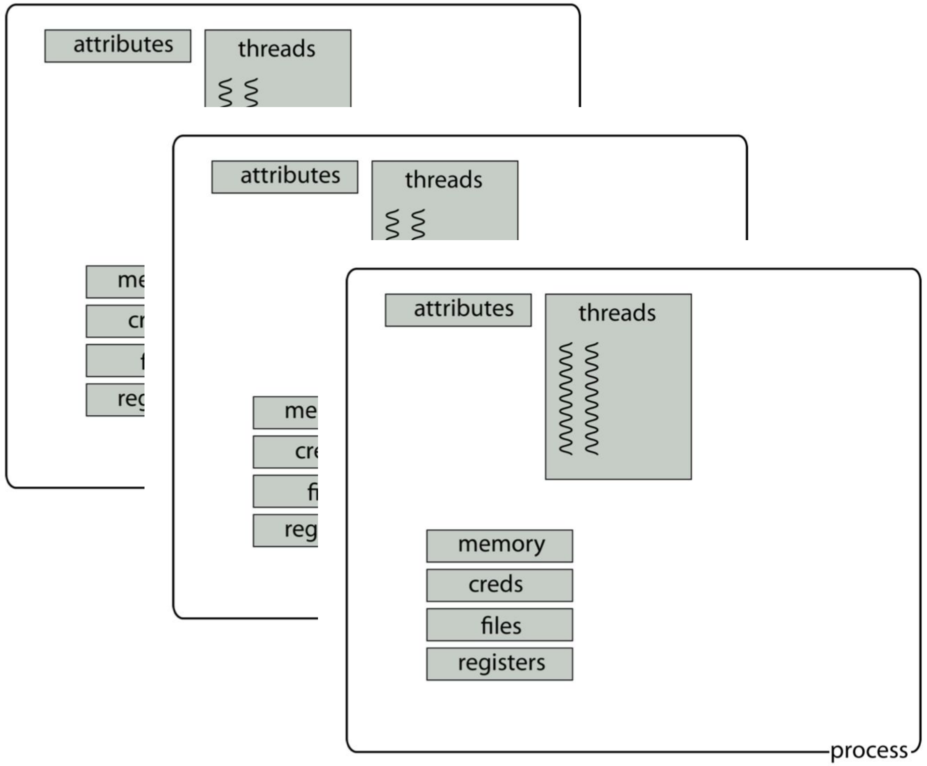
Process basic OS abstraction

- Unit of isolation, privilege separation, and program states
  - Threads for execution context
  - Other resources:
    - Memory
    - Credentials
    - Files
    - CPU register information
    - Attributes



Fork suffers from its own overhead

- Fork provides distinct isolation, privilege separation, and program state among processes

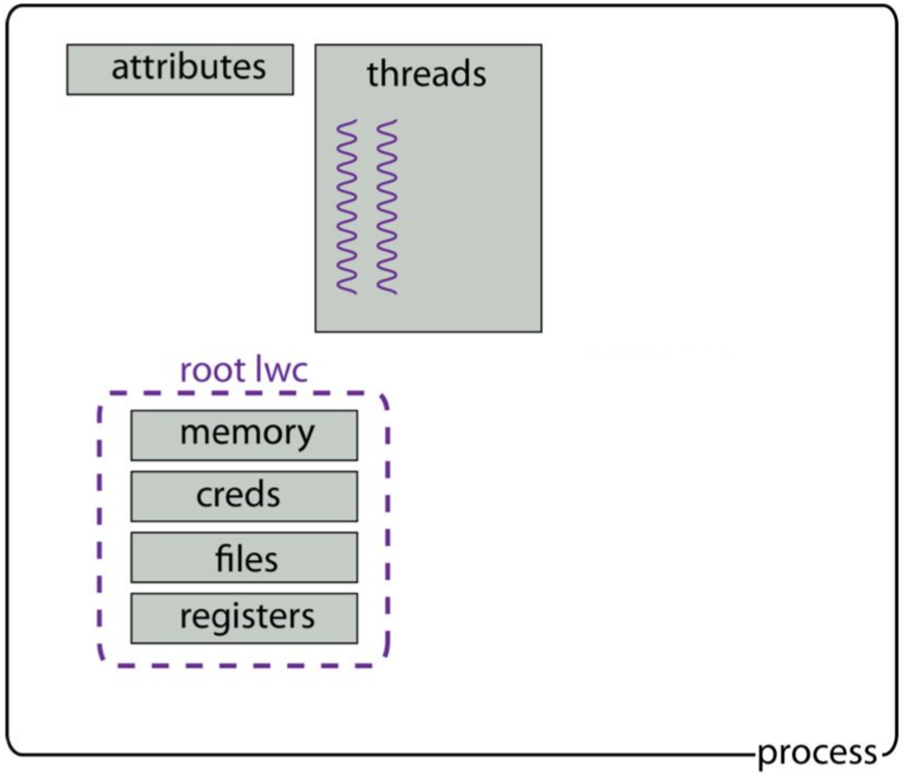


Cons:

- There is overhead on creating and scheduling processes

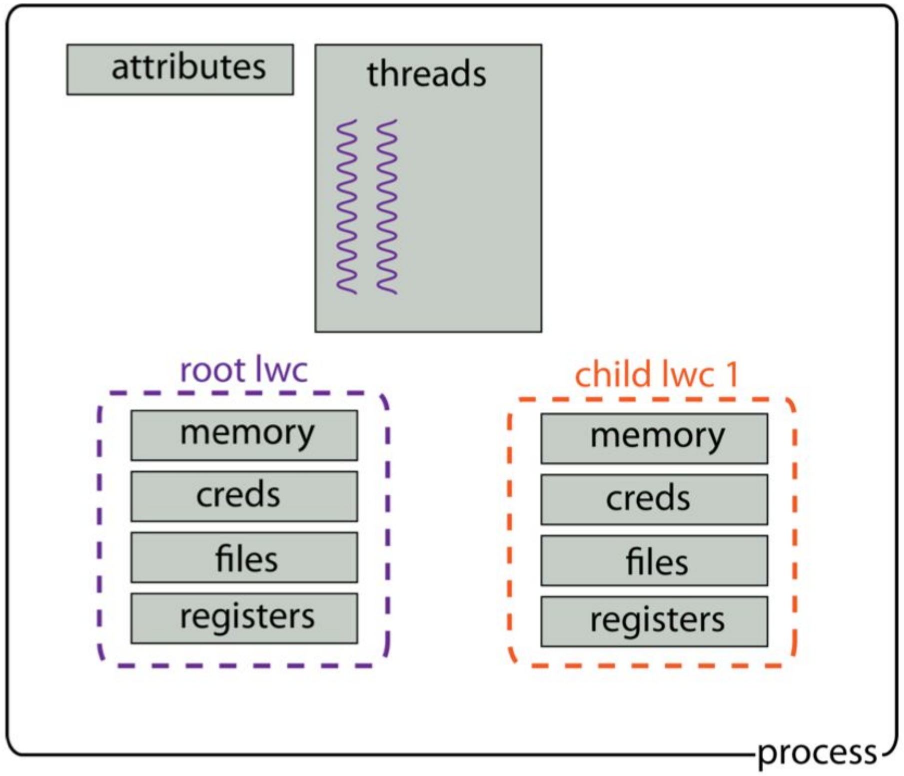
New isolation abstraction: Light-weight contexts

- Provides intra-process isolation of system resources
  - Memory, credentials, files, CPU registers



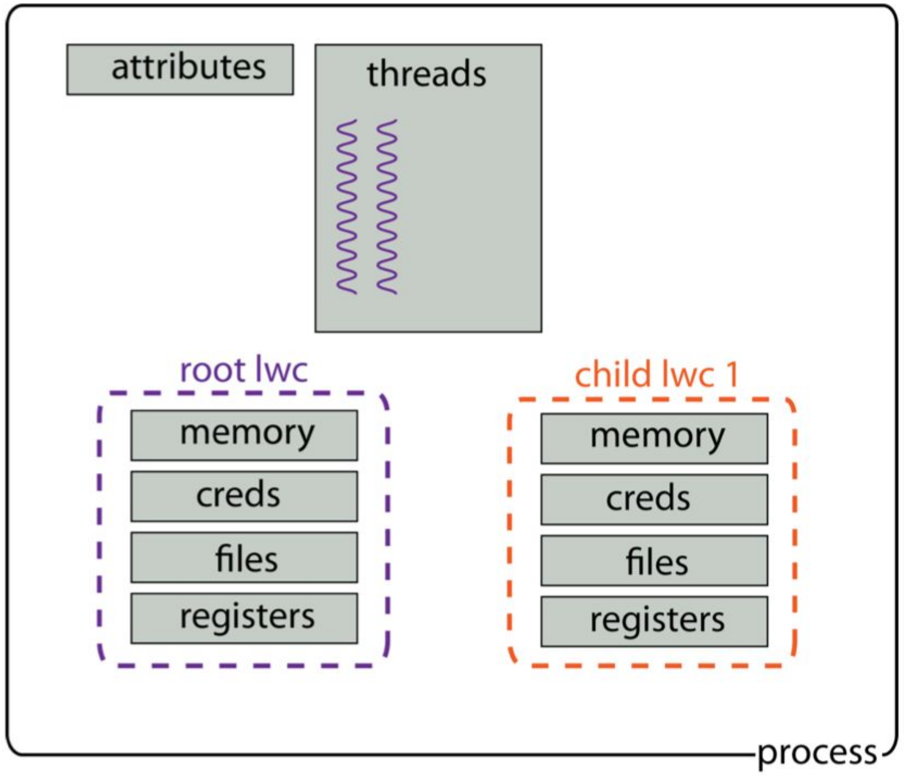
New isolation abstraction: Light-weight contexts

- Provides intra-process isolation of system resources
  - Memory, credentials, files, CPU registers



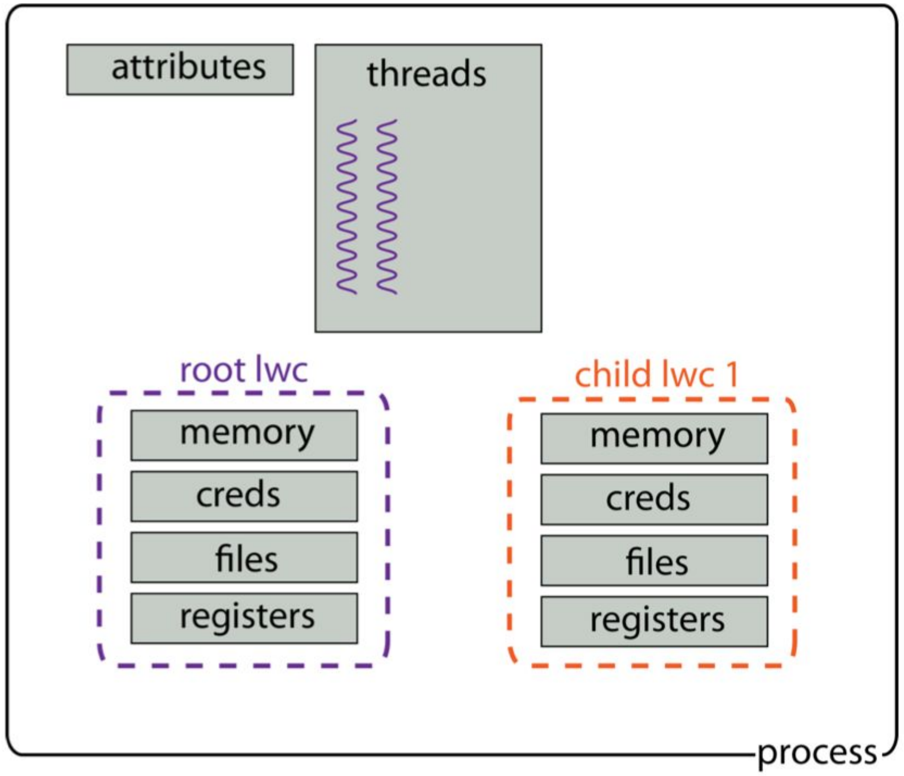
New isolation abstraction: Light-weight contexts

- Own virtual address space, page-mappings, file descriptors and credentials
- Process starts with a root lwC: can create more and give it whatever it wants to give
- Referenced by descriptor; may have multiple such descriptors
- Terminates when last reference goes away
- Creating an lwC does not start "running" it: it's just a context
  - When a thread switches to it, it copies the thread state and starts running.
- Switching lwCs is akin to a coroutine yield.



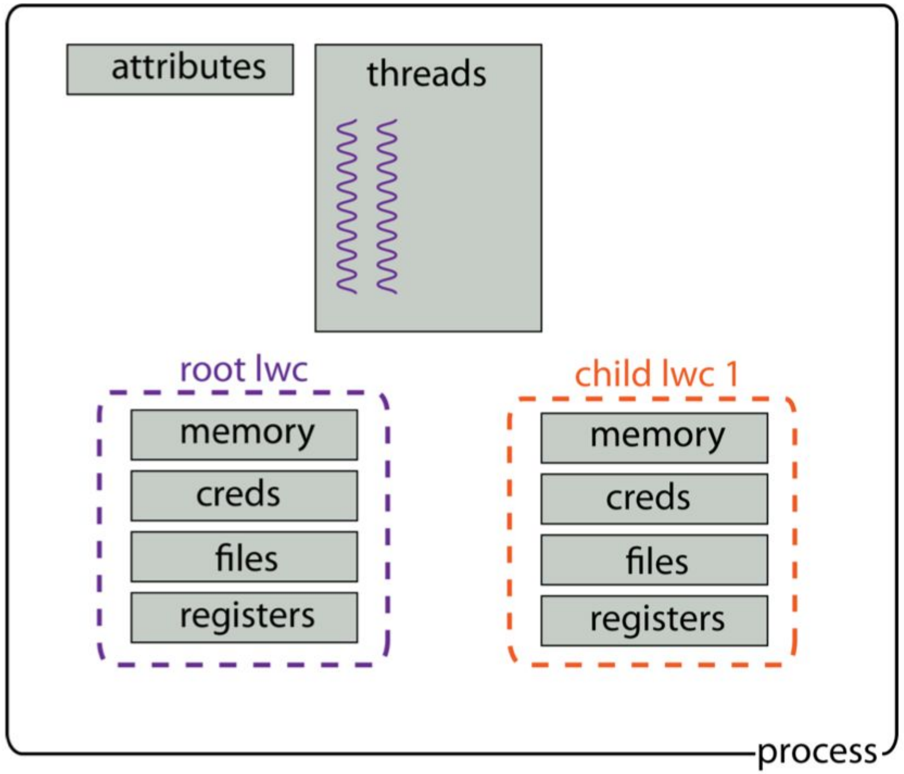
New isolation abstraction: Light-weight contexts

- Creating a child is kind of like clone: the parent gets to decide how resources are/are not shared with the child (using resource-spec)
- When a thread leaves an lwC, its state is remains in the original lwC, so that when it returns, it picks them up via arguments, as if it just made a switch into that lwC
- Capability-based system
- When you create an lwC, each descriptor can be COW, SHARED, or UNMAP.
- If allowed, you can map resources from one lwC into another using lwOverlay API



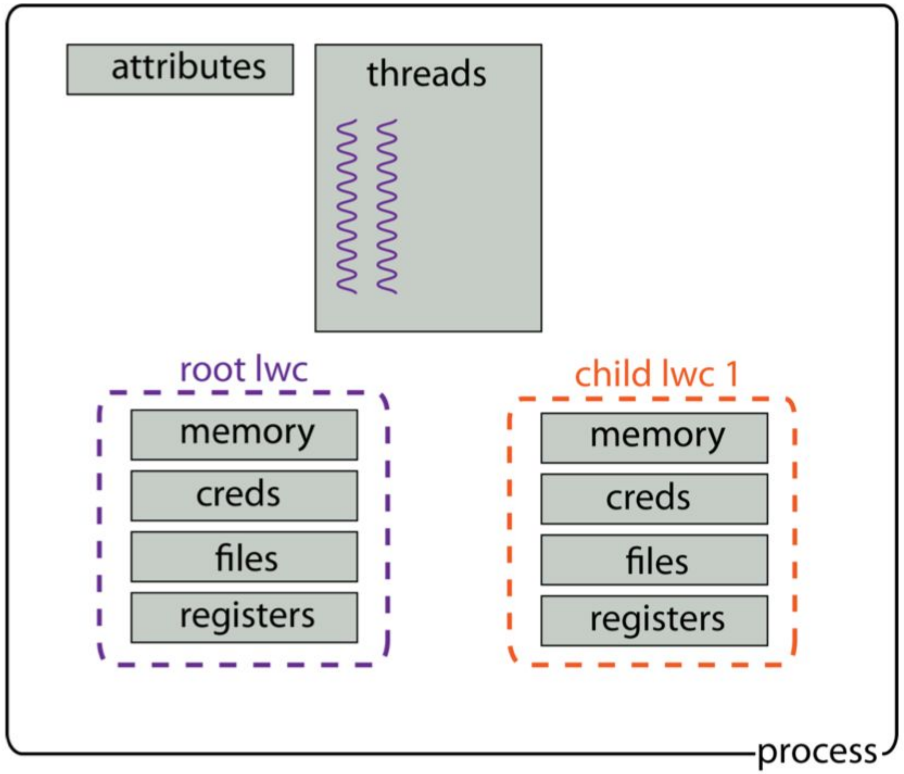
New isolation abstraction: Light-weight contexts

Create lwC		
new,caller,args	←	lwCreate(spec, options)
Switch to lwC		
caller,args	←	lwSwitch(target, args)
Resource Access		
status	←	lwRestrict(lwc, spec)
status	←	lwOverlay(lwc, spec)
status	←	lwSyscall(target, mask, syscall, args)



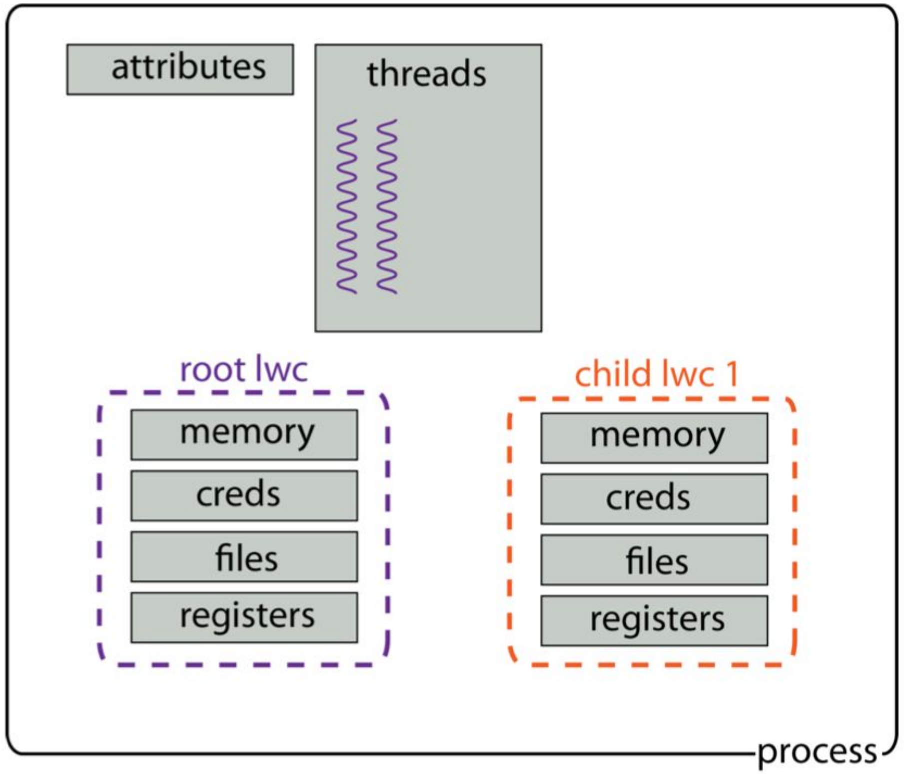
New isolation abstraction: Light-weight contexts

- Common use cases:
  - Snapshot
    - Create a context to save initial state (the child holds the saved state)
    - Handle a request
    - Now use the new context, which in turn destroys the original one, creates a new one
    - Lather, rinse, repeat



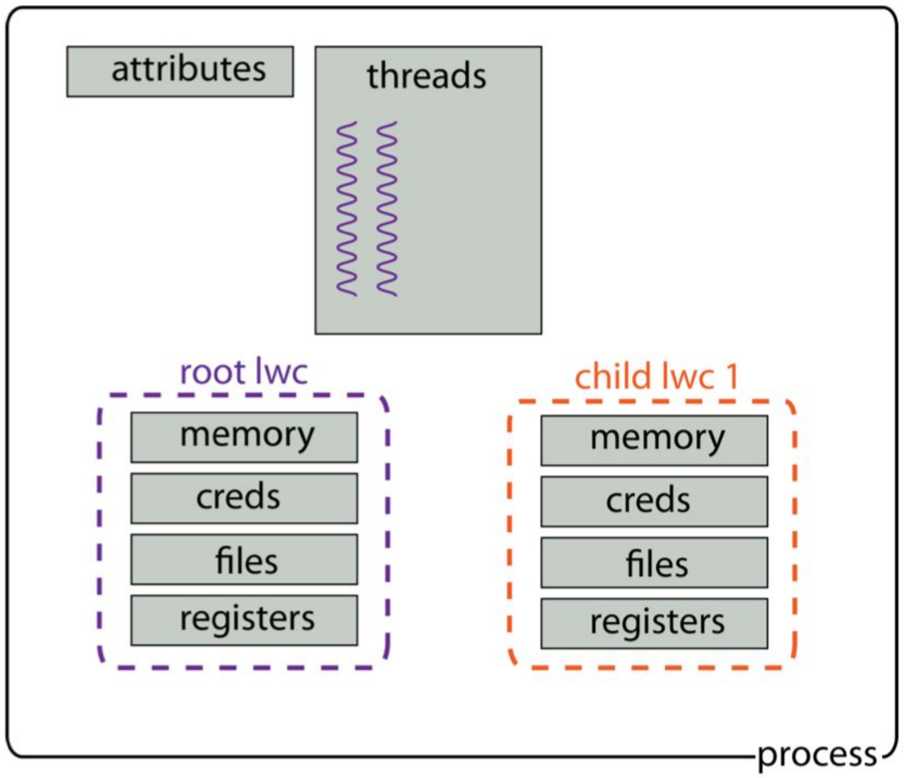
New isolation abstraction: Light-weight contexts

- Common use cases:
  - Server event-handling isolation (prevent information in different sessions from leaking to one another)
    - Server creates a socket descriptor for each client.
    - Uses different contexts to respond to each descriptor



New isolation abstraction: Light-weight contexts

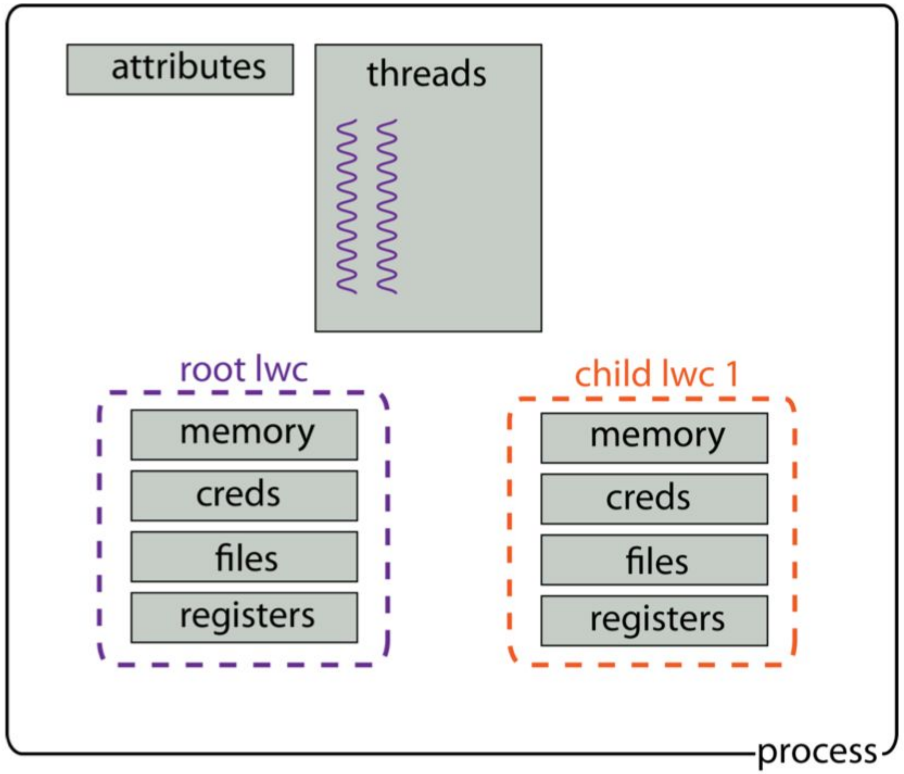
- Common use cases:
  - Isolation of sensitive data (e.g., a signing key)
    - Create a child who will have full rights to the key
    - Parent relinquishes access to child's space
    - Child enters infinite loop and everything gets a thread assigned maps in an argument buffer, signs it and unmaps it



Create lwC		
new,caller,args	←	lwCreate(spec, options)
Switch to lwC		
caller,args	←	lwSwitch(target, args)
Resource Access		
status	←	lwRestrict(lwc, spec)
status	←	lwOverlay(lwc, spec)
status	←	lwSyscall(target, mask, syscall, args)

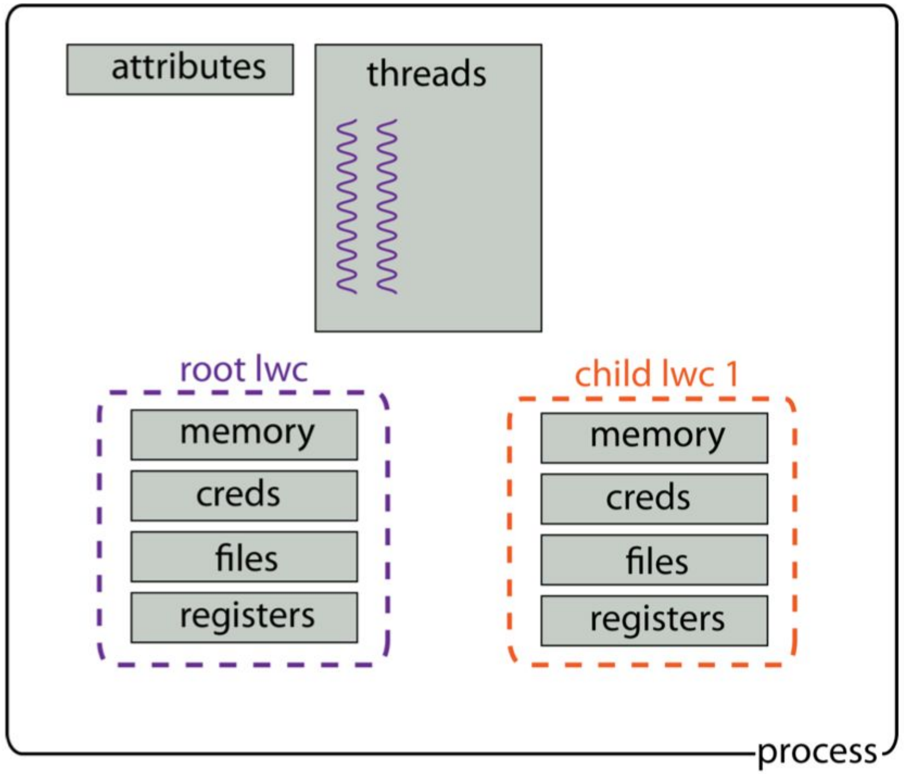
New isolation abstraction: Light-weight contexts

- Orthogonal to threads
- Allows snapshot/restore
- Privilege separation within a process
- No HW support
- No language/compiler support
- Focus on privilege, not resources
- No dynamic, runtime checking



New isolation abstraction: Light-weight contexts

- Evaluation:
  - Creation / destruction cost
    - No pages dirtied: 87.7 microseconds
  - Additional COW cost per page dirtied
    - 3.4 microseconds
  - Switching cost (between):
    - lWC: ~2 microsecond
    - Process: ~4.3 microsecond
    - k-thread: ~4.2 microsecond

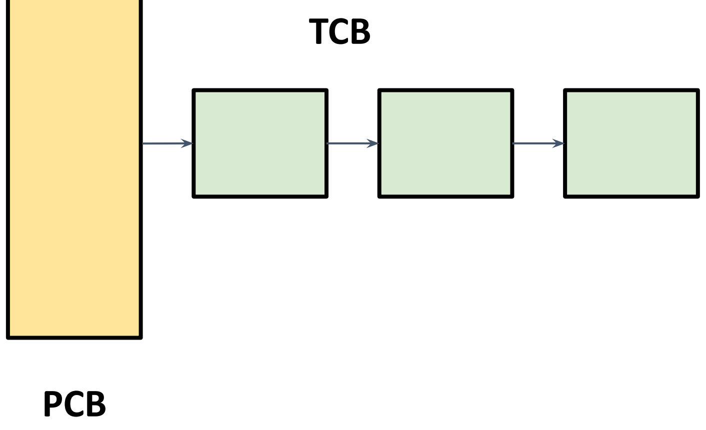


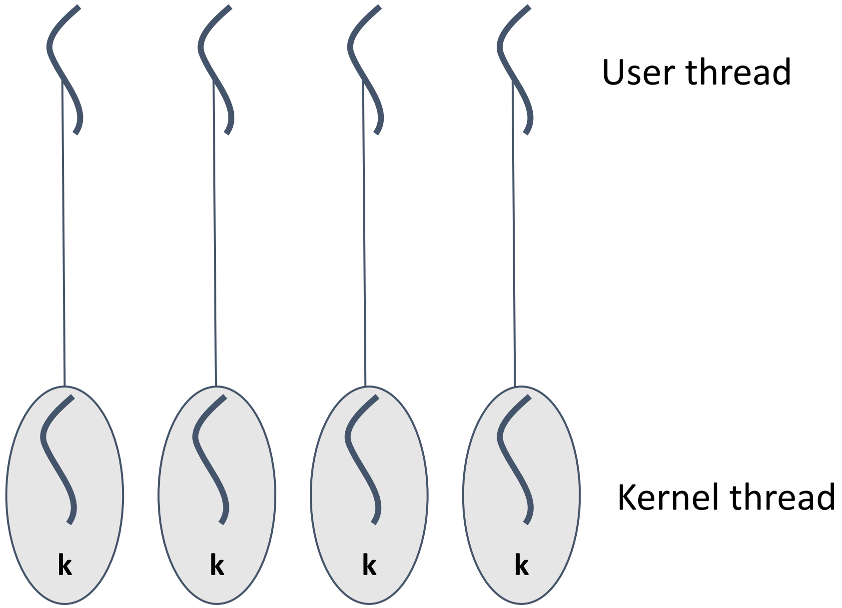
Focus of today's lecture

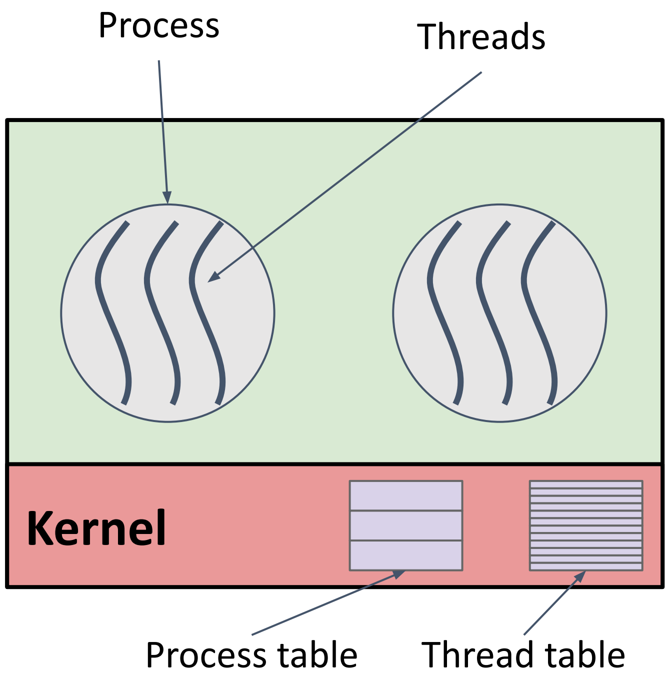
- Process abstraction
- Thread abstraction
- Process management in Linux
- Another OS abstraction: Lightweight contexts
- **Scheduler activations**

Kernel-level threads: Implementation

- Every thread operations are system calls
- 1:1 model





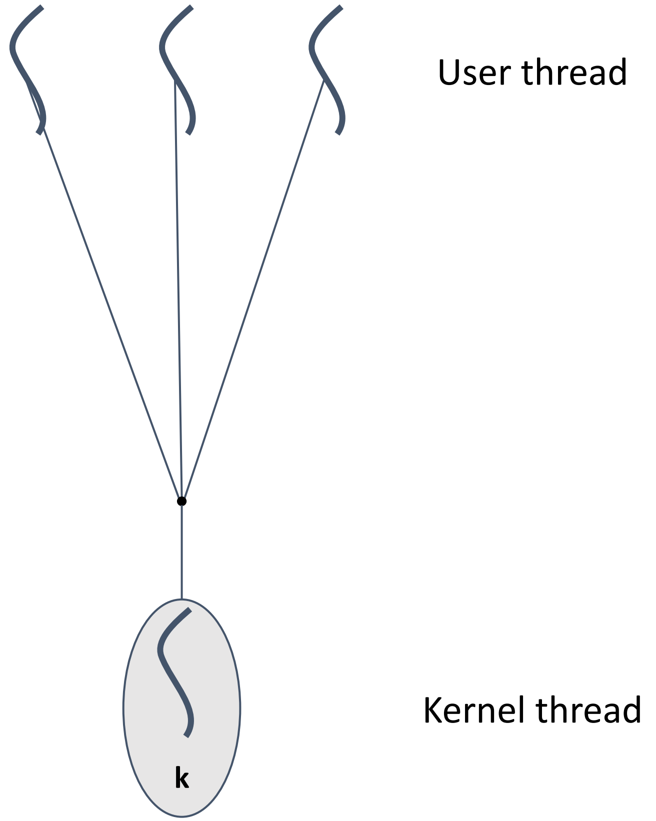


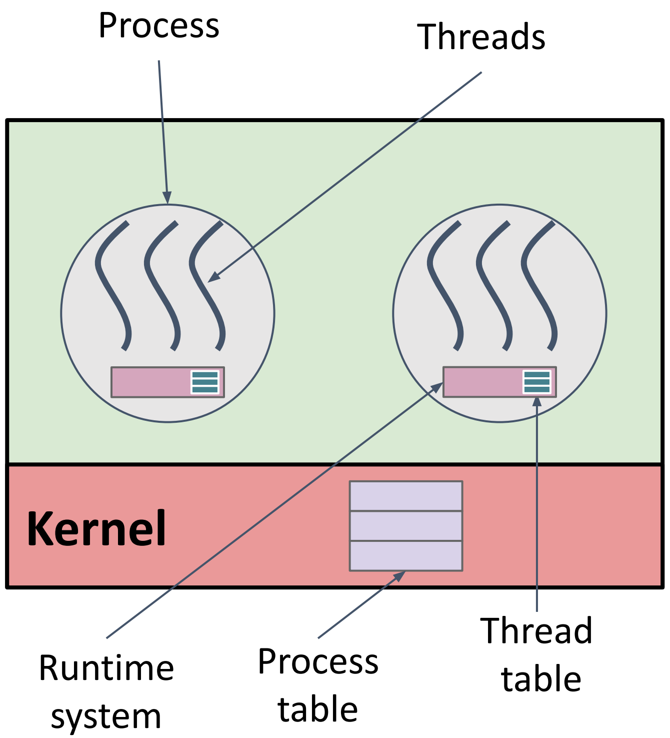
Kernel-level threads

- Pros
  - Cheaper than processes
  - Scheduling / management done by the kernel
    - Possible to overlap IO with computation
    - Can exploit multiple CPUs
- Cons
  - Still too expensive (compared to user-level threads)
  - Thread state in the kernel
  - Need to be general to support the needs of all programmers, languages, runtimes, etc.

User-level threads: Implementation

- Managed by runtime library
- Views each process as a "virtual processor"
- N:1 model





User-level threads

- Pros
  - Fast
  - Portable
  - Flexible
- Cons
  - Invisible to the OS; OS can make poor decisions
  - Cannot exploit multiple CPUs

Operation	FastThreads (User-level)	Topaz threads (Kernel-level)	Ultrix processes
Null Fork	34µs	948µs	11300µs
Signal-Wait	37µs	441µs	1840µs

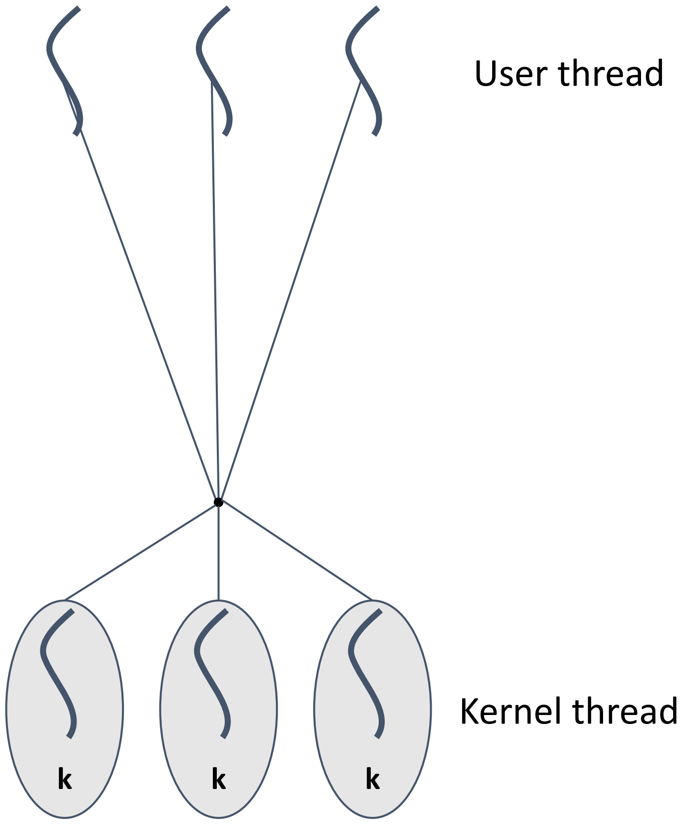
Goals

- The performance and flexibility of user-level threads
  - Performance of user-level systems in the common case
  - Simplify application-specific customization:
    - Scheduling policy, concurrency models, etc.
- The functionality of kernel threads
  - No processor idles in the presence of ready threads
  - No high priority thread waits for a processor while a low-priority thread runs
  - Thread traps to the kernel to block, the processor can be used to run another thread from the same or from a different address space

A simple solution

- Use M:N threading model

- Problems:
    - Preemption of lock holder?
    - Scheduling an idle thread?
    - Preempting a high-priority thread?
    - Running out of kernel threads?



Observations

- Kernel threads are the wrong abstraction for supporting user-level thread management
- The kernel needs access to user-level scheduling information
- The user-level thread system needs to be aware of kernel events

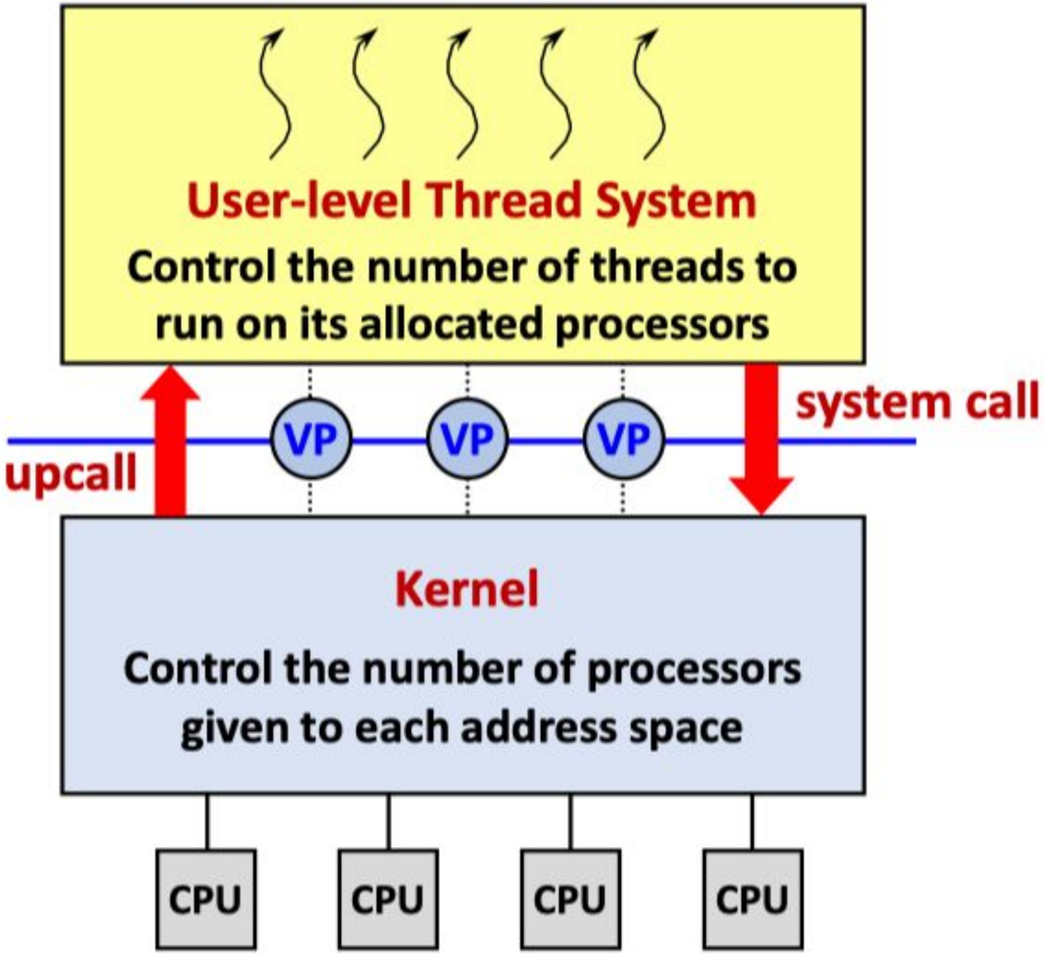
Scheduler activation

- Serves as a vessel, or execution context, for running user-level threads (as an extension of a kernel thread)
- Notifies the user-level thread system of a kernel event via ***upcall***
- Requires two stacks:
  - A kernel-level stack: used for system calls
  - A user-level stack: used for upcalls
  - Note: Each user-level thread has its own stack
- Activation control block
  - Saves the processor context of the activation current user-level thread when the thread is stopped by the kernel

Scheduler activation

- Serves as a vessel, or execution context, for running user-level threads (as an extension of a kernel thread)
- Notifies the user-level thread system of a kernel event via ***upcall***
- Requires two stacks:
  - A kernel-level stack: used for system calls
  - A user-level stack: used for upcalls
  - Note: Each user-level thread has its own stack
- **Activation control block**
  - Saves the processor context of the activation current user-level thread when the thread is stopped by the kernel

Scheduler activation: Overview

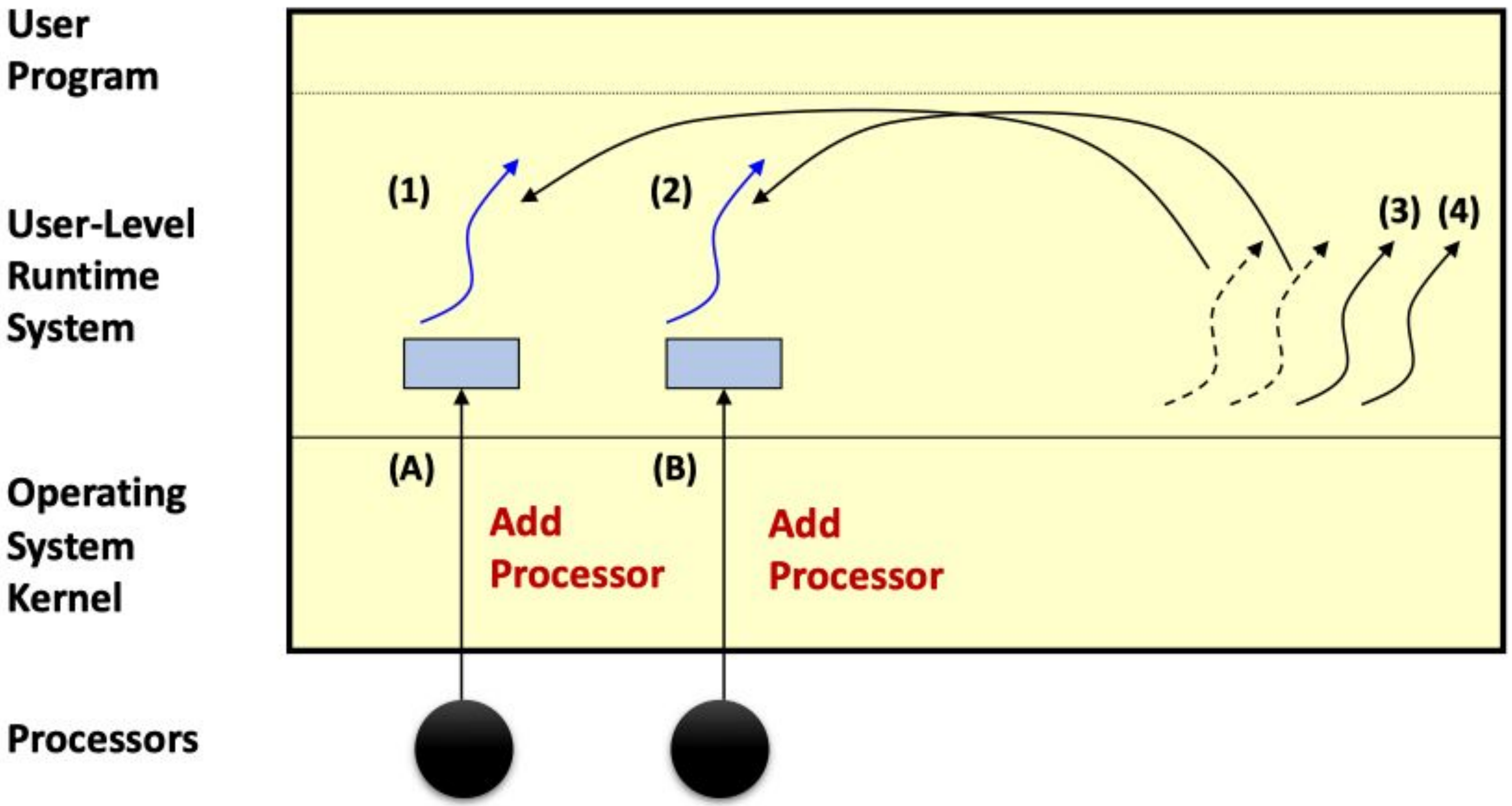


Notifies the kernel when the application needs more or fewer processors

Notifies the user-level thread system whenever the kernel changes the number of processors assigned to it

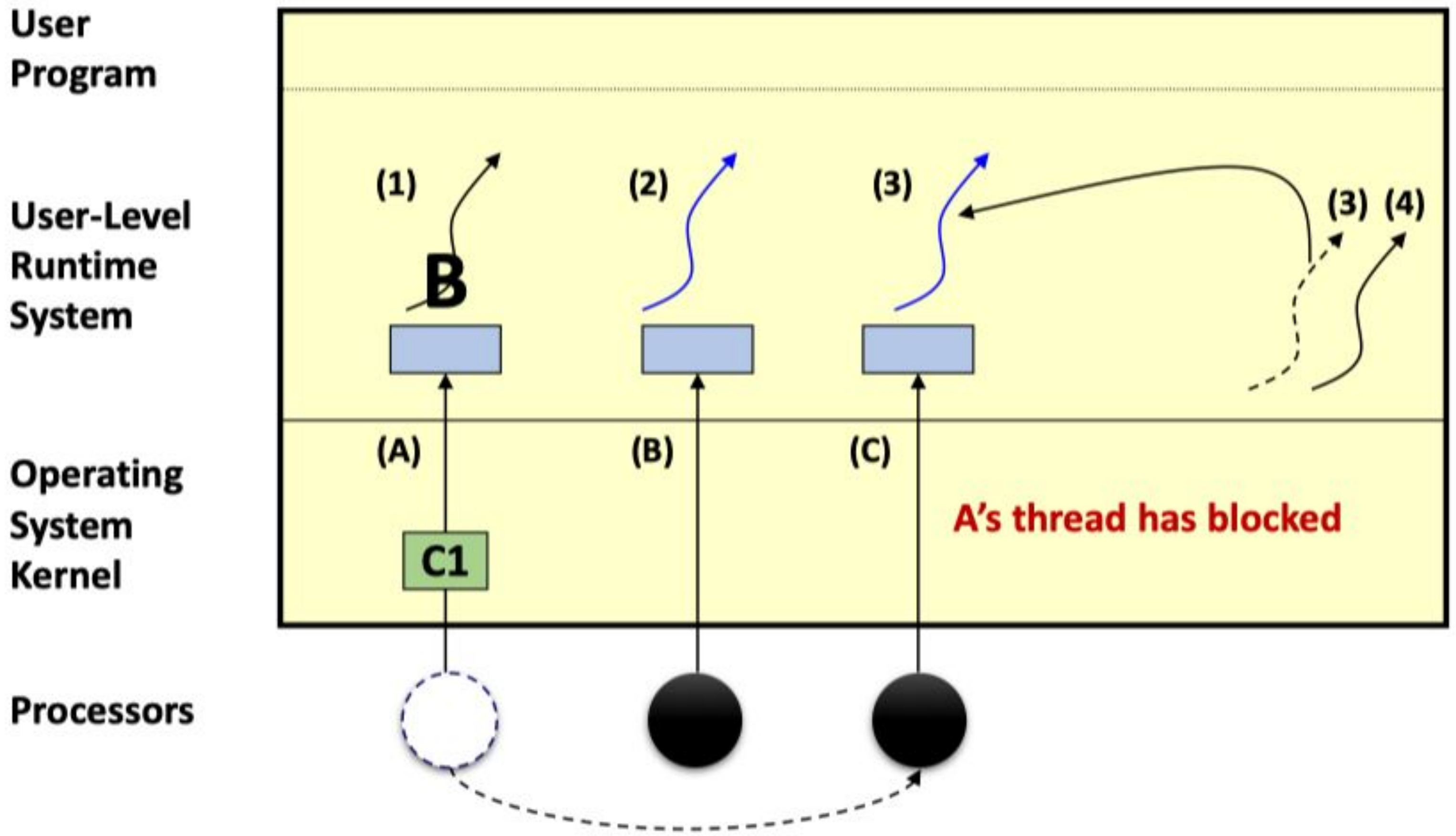
Example

- T1: the kernel allocates two processors (two kernel threads)



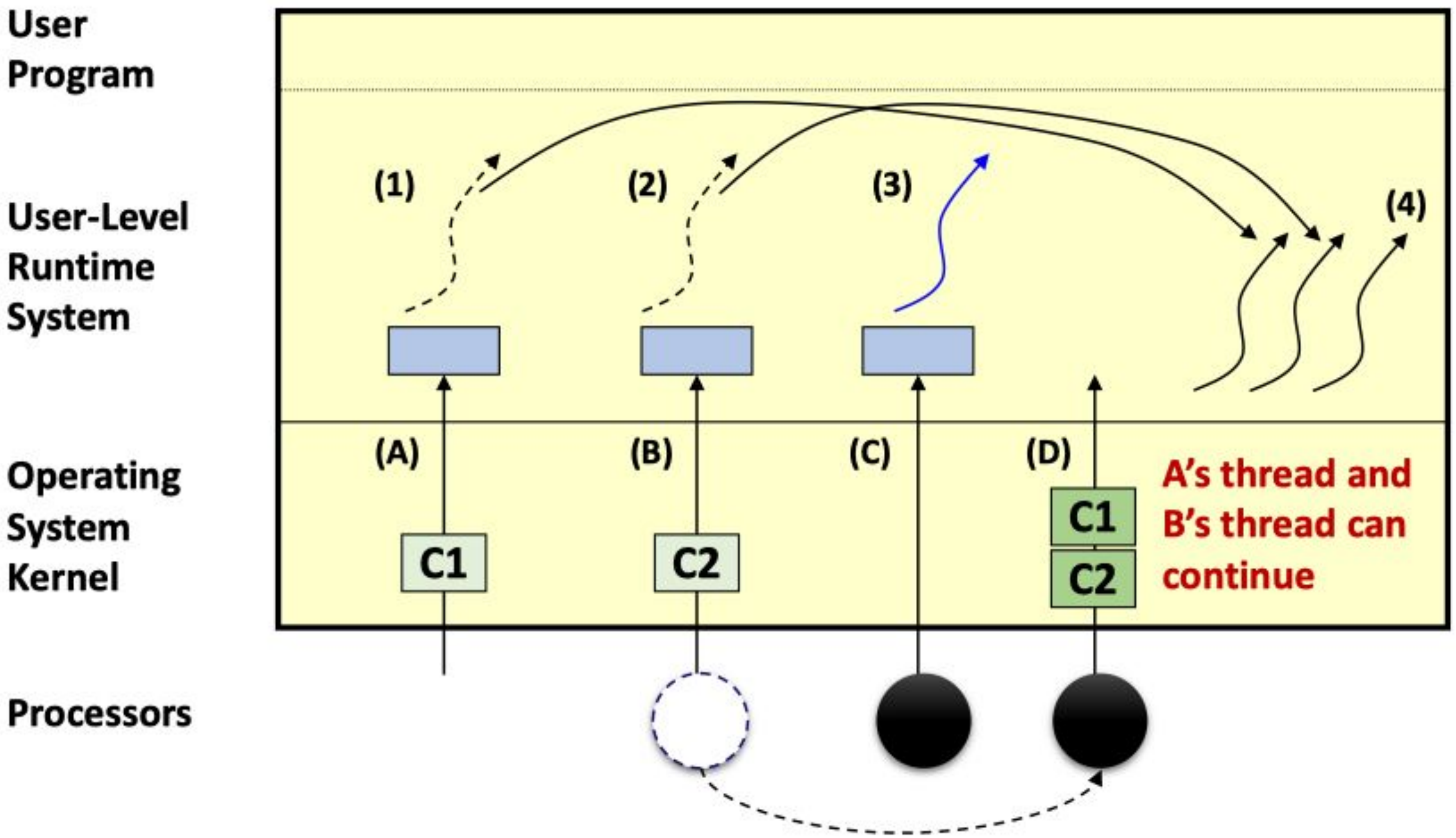
Example

- T2: Thread 1 blocks in the kernel for IO



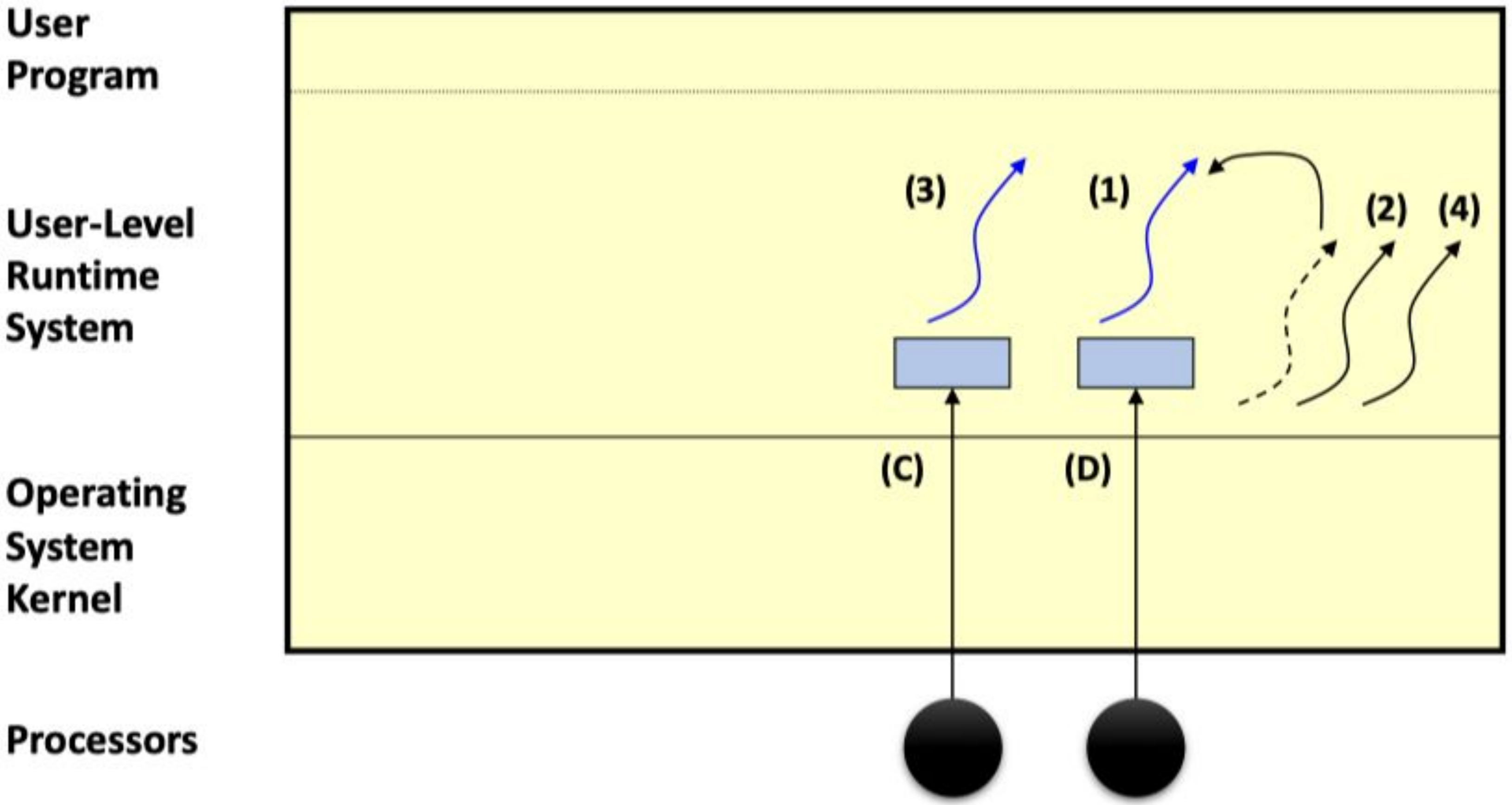
Example

- T3: Thread 1 completes the IO



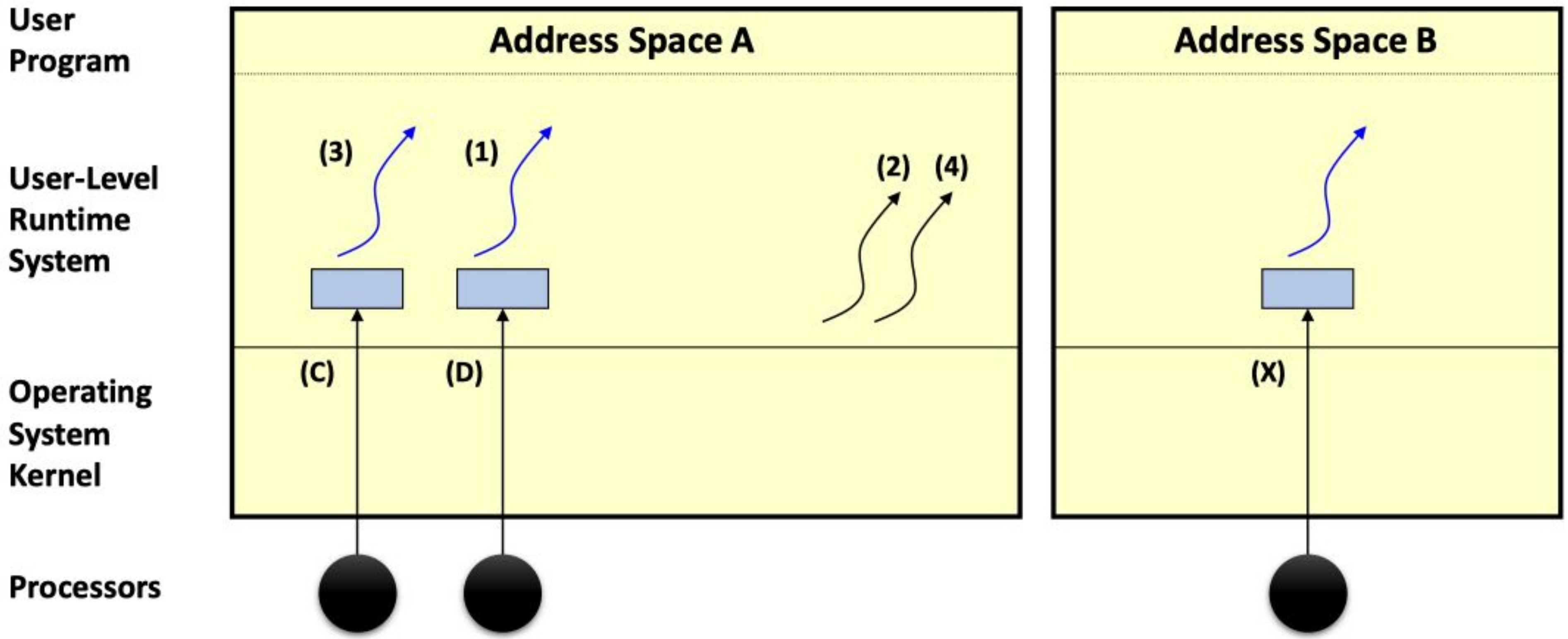
Example

- T4: Thread 1 resumes



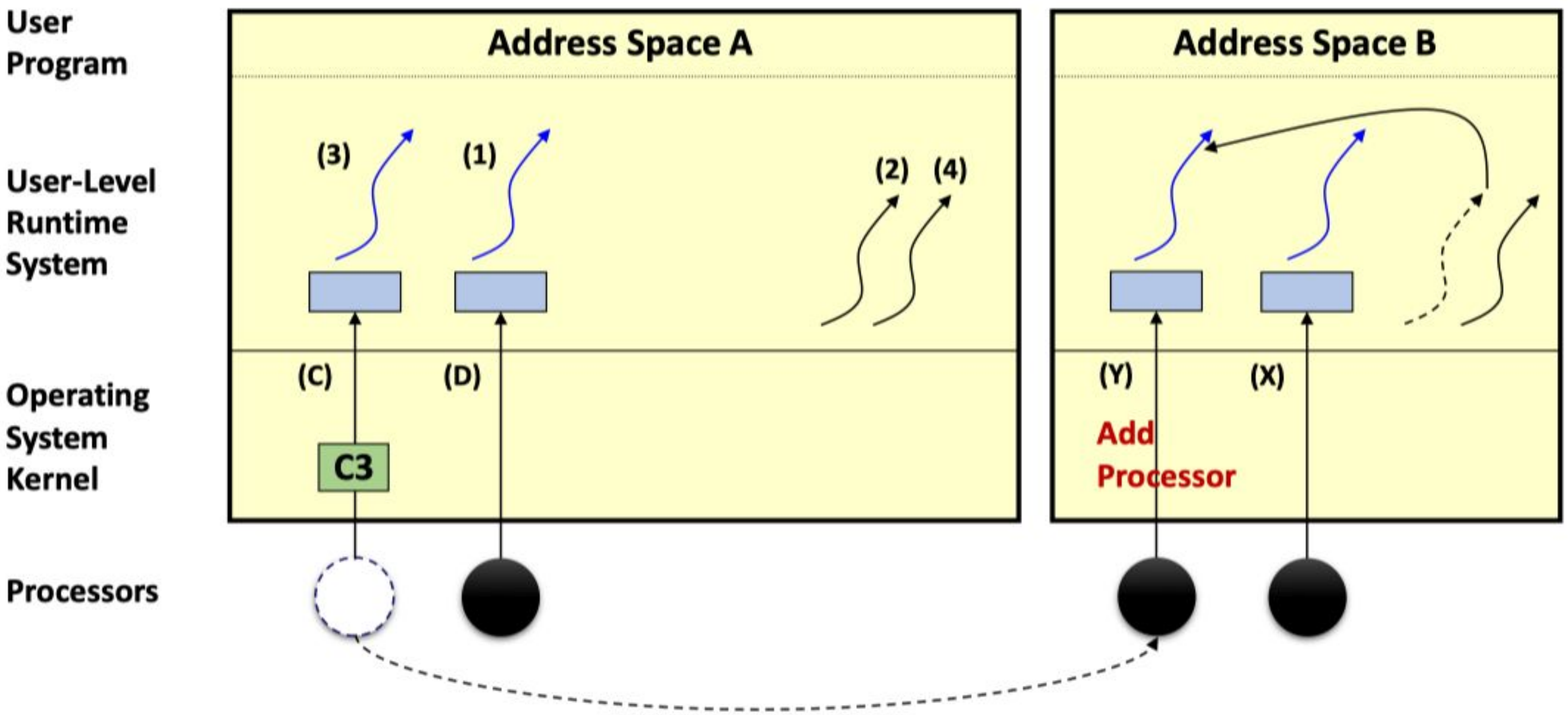
Example

- T5: Kernel wants to take a processor (kernel thread) away from address space A



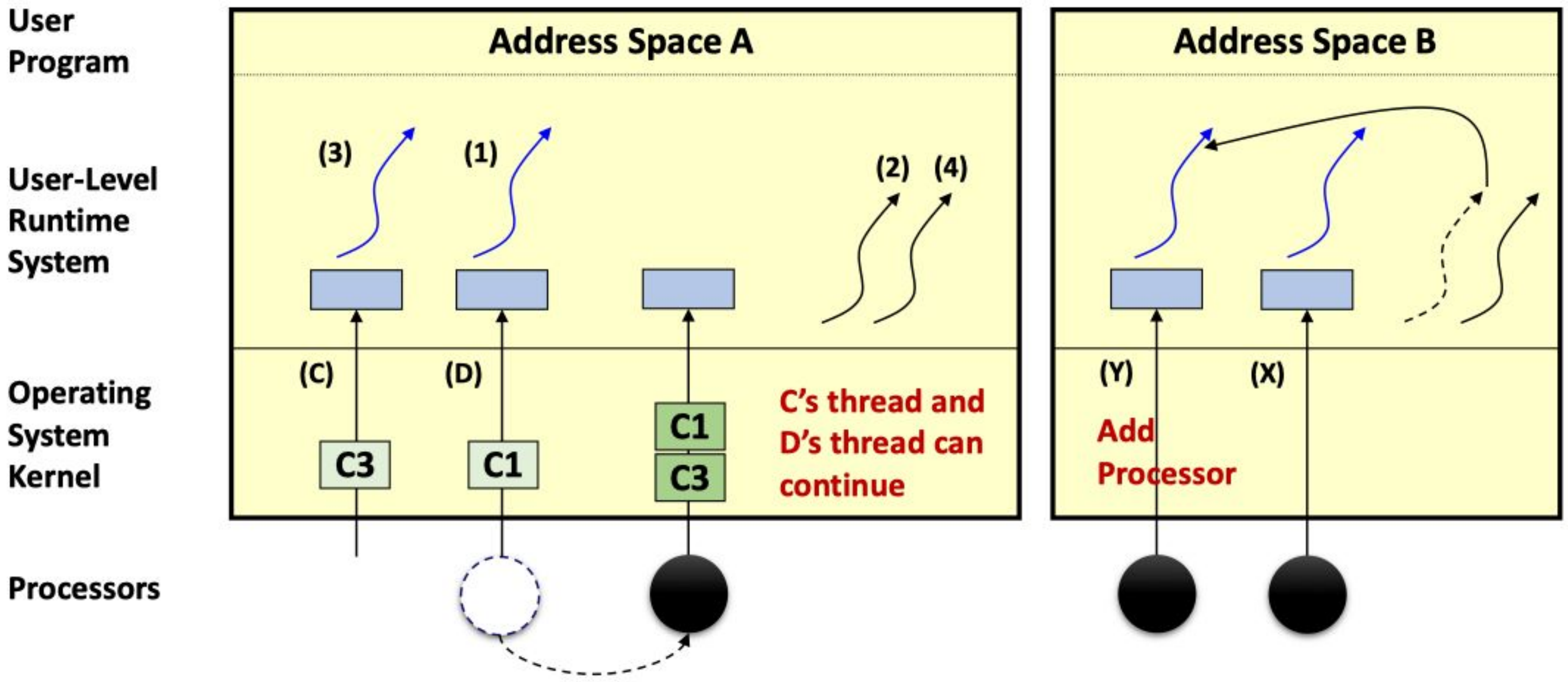
Example

- T6: Thread 3 is preempted and the processor is allocated to B



Example

- T7: Thread 1 is preempted and kernel notifies to A



Upcall: kernel → user

- Add this processor (processor #)
  - Execute a runnable user-level thread
- Processor has been preempted (preempted SA# and its machine state)
  - Return to the ready list of that user-level in the context of preempted SA
- Scheduler activation has been blocked (blocked SA#)
  - The blocked SA is no longer using its processor
- Scheduler activation has unblocked (unblocked SA# and its machine state)
  - Return to the ready list the user-level thread was executing in the context of the blocked SA

System call: user → kernel

- Add more processors
  - Allocate processors along with the SAs
- The processor is idle
  - Preempt this processor if another address space needs it
- The user-level thread system need not tell the kernel about every thread operation

Critical sections

- What to do if a preempted or blocked thread is in the critical section?
  - Poor performance or deadlock
- Solution based on “recovery”:
  - Check whether the preempted thread was in a critical section
  - If so, it is continued temporarily via a user-level context switch
- Performance enhancement
  - Make a copy of each critical section
  - Runtime checks using the section begin/end address
  - Normal execution uses the original version
  - The copy returns to the scheduler at the end of the critical section
  - Imposes no overhead in the common case

Application transparency

- The application can build any concurrency model
- The kernel needs no knowledge of the data structures used to represent parallelism at the user level

- Scheduler activations provide a “mechanism”, not a “policy”

Basic performance

Operation	FastThreads on Topaz threads	FastThreads on Scheduler Activations	Topaz threads (Kernel-level)	Ultrix processes
Null Fork	34µs	37µs	948µs	11300µs
Signal-Wait	37µs	42µs	441µs	1840µs

- Performance is slightly worse due to the cost:
  - Maintaining a counter for number of busy threads
  - Cost of checking whether a preempted thread is being resumed
- Upcall can contribute up to 5x slowdown
  - Dynamic allocation/deallocation of SA and management of threads in user space

Issues with scheduler activation (Solaris 2)

- M:N model is too complex:
  - Signal handling
  - Automatic concurrency management
  - Poor scalability due to synchronization in the user-level thread
  - Advances in kernel thread scalability

Summary

- Process: An abstraction for protection
- Thread: An execution context
- Various process models: 1:1, N:1, M:N
- Intra-process isolation is also possible with lWC:
  - Follow the same model of kernel resources entirely in the userspace
- Scheduler activation gives hints to the user space to make better scheduling / execution decisions