Threads Systems Concepts

Review: Why Threads?

- Processes do not share resources very well
  - Why?
- Process context switching cost is very high
  - Why?

↓

- Threads: light-weight processes

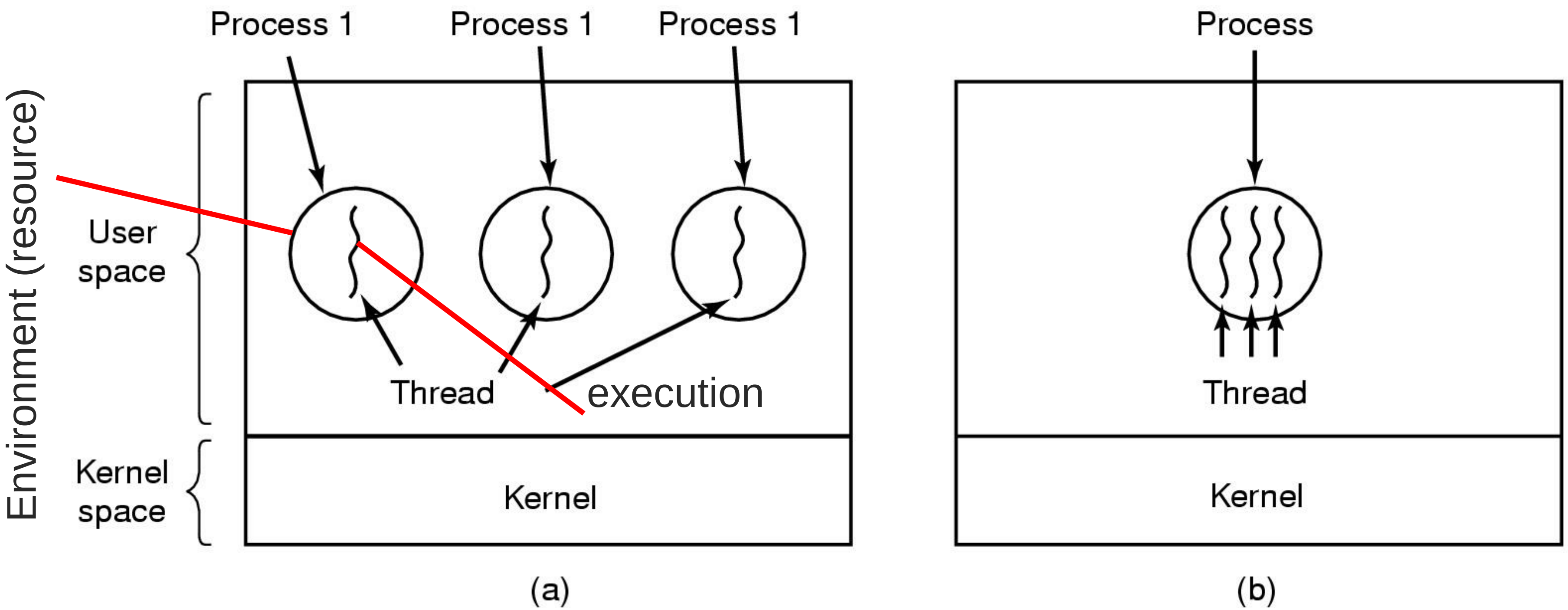
Benefits of Threads

- Takes less time
  - To create a new thread
  - To terminate a thread
  - To switch between two threads
- Inter-thread communication without invoking the kernel

We like our Threads …

- Foreground and background work
- Asynchronous processing
- Speed of execution
- Modular program structure

Threads: Lightweight Processes



a) Three processes each with one thread
b) One process with three threads

Tasks Suitable for Threading

- Has multiple parallel sub-tasks
- Some sub-tasks block for potentially long waits
- Some sub-tasks use many CPU cycles
- Must respond to asynchronous events

Questions

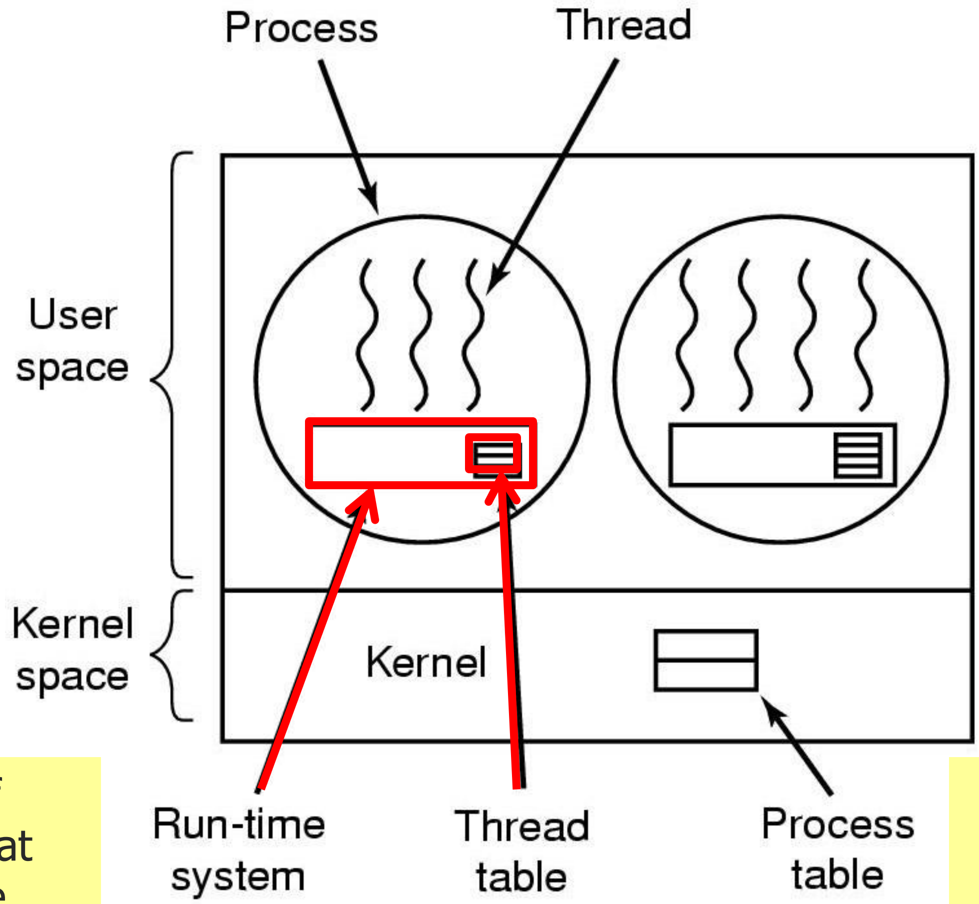
- What are the similarities between processes and threads?

- What are the differences between processes and threads?

Thread Packages

- Kernel thread packages
  - Implemented and supported at kernel level
- User-level thread packages
  - Implemented at user level
  - Kernel perspective: everything is a single-threaded process

Threads in User Space (Old Linux)



Collection of procedures that manages the threads

Keep track of threads in process (analogous to kernel process table)

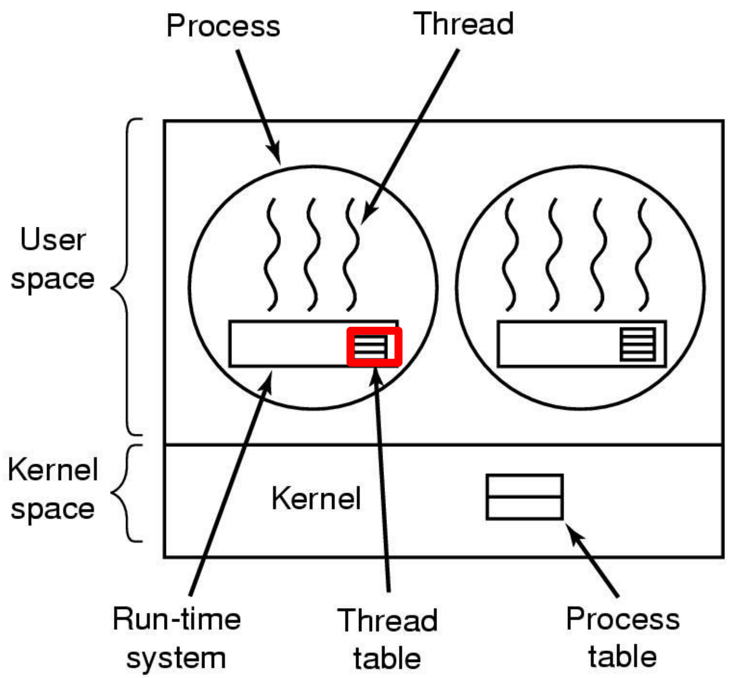
User-level Threads

- Advantages
  - Fast Context Switching: keeps the OS out of it!
    - User level thread libraries do not require system calls
      - No call to OS and no interrupts to kernel
    - thread_yield
      - Save the thread information in the thread table
      - Call the thread scheduler to pick another thread to run
    - Saving local thread state scheduling are local procedures
      - No trap to kernel, low context switch overhead, no memory switch
  - Customized Scheduling (at user level)

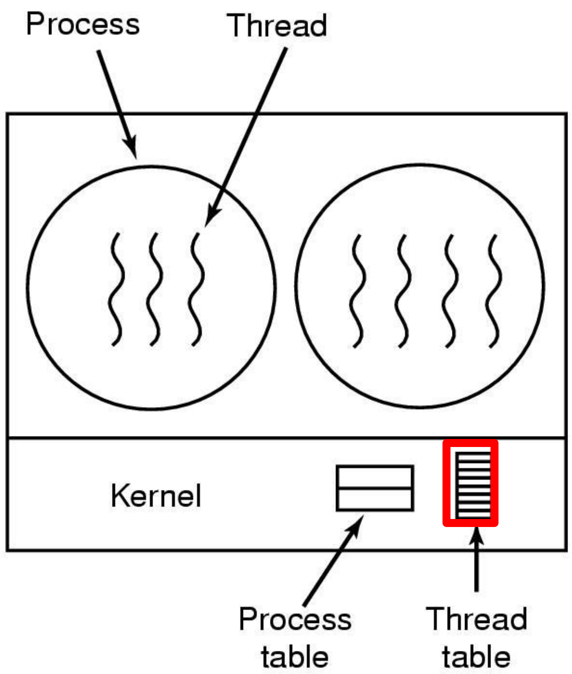
User-level Threads

- Disadvantages
  - What happens if one thread makes a blocking I/O call?
    - Change the system to be non-blocking
    - Always check to see if a system call will block
  - What happens if one thread never yields?
    - Introduce clocked interrupts
  - Multi-threaded programs frequently make system calls
    - Causes a trap into the kernel anyway!

Kernel Threads



User-level Threads



Kernel-level Threads

Kernel-level Threads

- Advantages
  - Kernel schedules threads in addition to processes
  - Multiple threads of a process can run simultaneously
    - Now what happens if one thread blocks on I/O?
    - Kernel-level threads can make blocking I/O calls without blocking other threads of same process
  - Good for multicore architectures

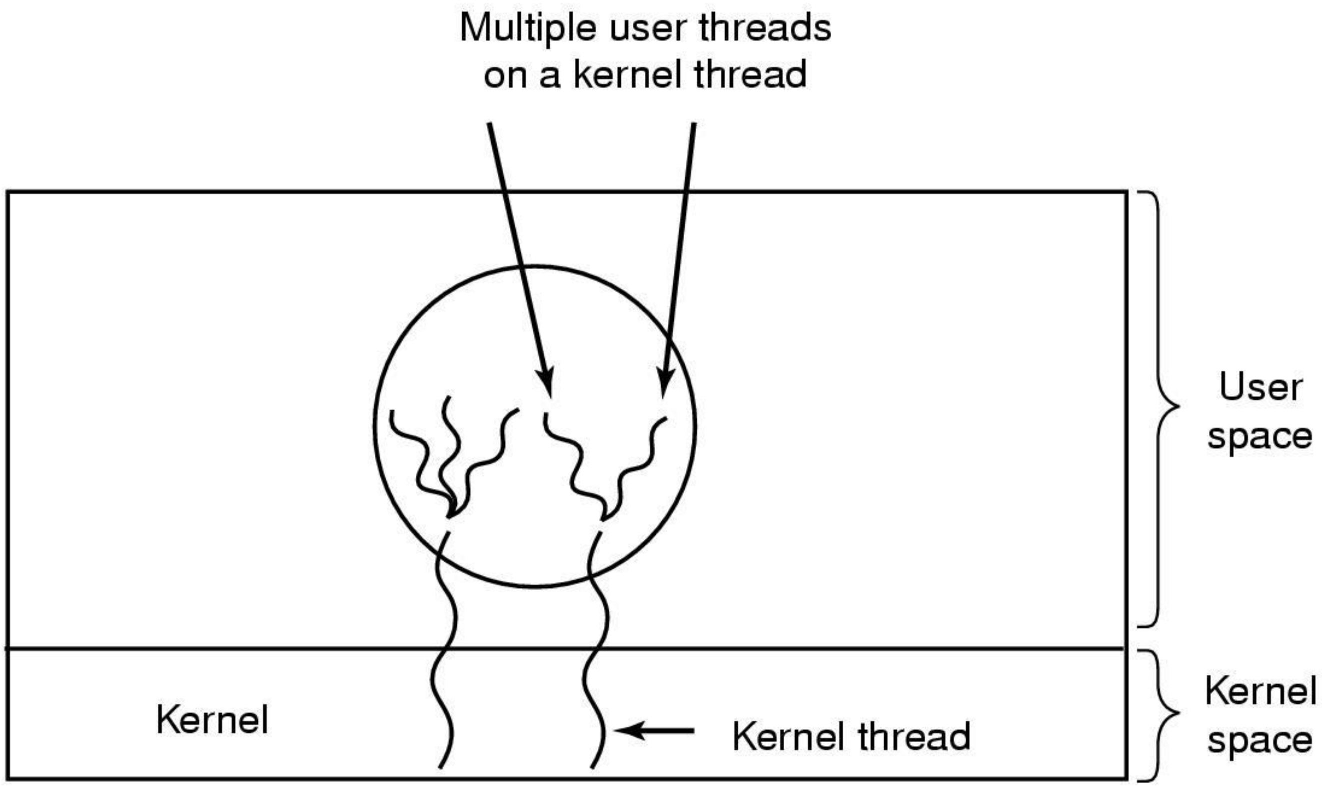
Kernel-level Threads

- Disadvantages
  - Overhead in the kernel… extra data structures, scheduling, etc.
  - Thread creation is expensive
    - Have a pool of waiting threads
  - What happens when a multi-threaded process calls **`fork()`**?
  - Which thread should receive a signal?

Trade-offs?

- Kernel thread packages
  - Each thread can make blocking I/O calls
  - Can run concurrently on multiple processors
- Threads in User-level
  - Fast context switch
  - Customized scheduling
  - No need for kernel support

Hybrid Implementations (Solaris)



Multiplexing user-level threads onto kernel-level threads

When can we add Concurrency?

- Work that can be executed, or data that can be operated on, by multiple tasks simultaneously
- Block for potentially long I/O waits
- Use many CPU cycles in some places but not others
- Must respond to asynchronous events
- Some work is more important than other work (priority interrupts)

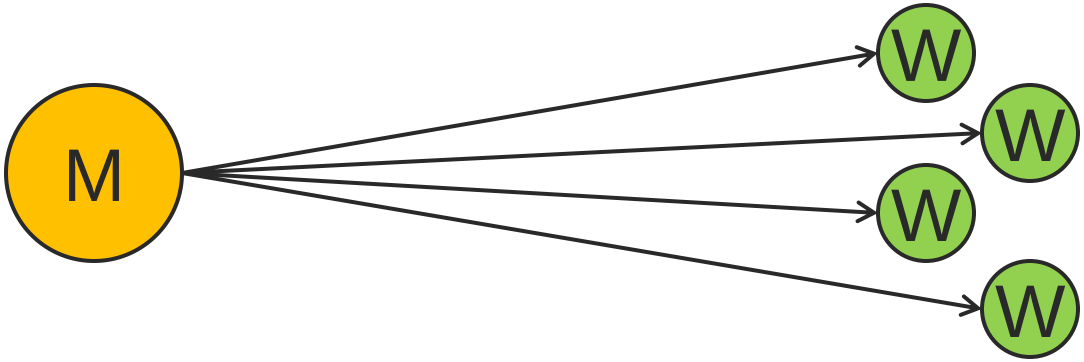
Concurrent Programming

- Assumptions
  - Two or more threads (or processes)
  - Each executes in (pseudo) parallel and can't predict exact running speeds
  - The threads can interact via access to a shared variable
- Example
  - One thread writes a variable
  - The other thread reads from the same variable
- Problem
  - The order of READs and WRITEs can make a difference!!!

Common Ways to Structure Multi-threaded Code

- Manager/worker
  - Single thread (manager) assigns work to other threads (workers)
  - Manager handles all input and parcels out work

Manager/Worker Model



Manager:

```
create N workers
forever {
   get a request
   pick free worker
}
```

Worker:

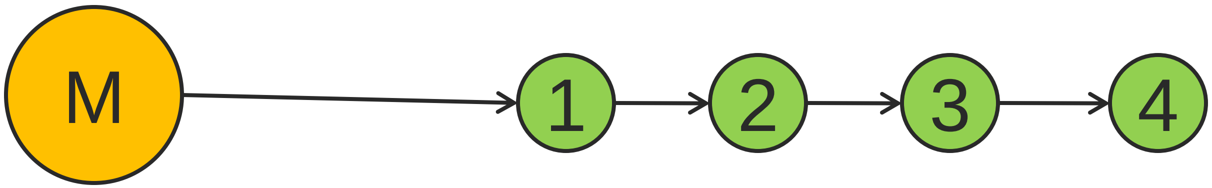
```
forever {
   wait for request
   perform task
}
```

- Challenges
  - Not enough/too many worker threads

Common Ways to Structure Multi-threaded Code

- Manager/worker
  - Single thread (manager) assigns work to other threads (workers)
  - Manager handles all input and parcels out work
- Pipeline
  - Task is broken into a series of sub-tasks
  - Each sub-task is handled by a different thread

Pipeline Model



Manager:

```
create N stages
forever {
   get a request
   pick 1st stage
}
```

Stage N:

```
forever {
   wait for request
   perform task
   pick stage n+1
}
```

- Challenges
  - Balancing per-stage load/parallelism

Common Ways to Structure Multi-threaded Code

- Manager/worker
  - Single thread (manager) assigns work to other threads (workers)
  - Manager handles all input and parcels out work
- Pipeline
  - Task is broken into a series of sub-tasks
  - Each sub-task is handled by a different thread
- Peer
  - Same structure as manager/worker model
  - After the main thread creates other threads, it participates in the work

Race Conditions

- What is a race condition?
  - Two or more threads have an inconsistent view of a shared memory region (i.e., a variable)
- Why do race conditions occur?
  - Values of memory locations replicated in registers during execution
  - Context switches at arbitrary times during execution
  - Threads can see “stale” memory values in registers

Remember this code?

```
int x = 1;
main(…) {
   pthread_t tid;
   pthread_create(
      &tid,NULL,
      func,NULL);
   func(NULL);
   x = x + 1;
}
```

```
void* func(void*p){
   x = x + 1;
   printf("x is
      %d\n");
   return NULL;
}
```

What is the output?

Race Conditions

- Race condition
  - Whenever the output depends on the precise execution order of the processes!!!
- What solutions can we apply?
  - Prevent context switches by preventing interrupts
  - Make threads coordinate with each other to ensure mutual exclusion in accessing critical sections of code

Threading Pitfalls

- Global variables
  - No protection between threads
    - Disallow all global variables
    - Introduce new thread-specific global variables
    - Introduce new library functions
- Are my libraries thread-safe?
  - May use local variables
  - May not be designed to be interrupted
    - Create wrappers

Threadssafe Library Calls

```c
#include <string.h>

char *token;
char *line = "LINE TO BE SEPARATED";
char *search = " ";

/* Token will point to "LINE". */
token = strtok(line, search);

/* Token will point to "TO". */
token = strtok(NULL, search);
```

```c
#include <string.h>

char *token;
char *line = "LINE TO BE SEPARATED";
char *search = " ";

/* Token will point to "LINE". */
token = strtok_r(line, search);

/* Token will point to "TO". */
token = strtok_r(NULL, search);
```

Threadssafe Library Calls

```c
#include <string.h>

char *token;
char *line = "LINE TO BE SEPARATED";
char *search = " ";
char *state;

/* Token will point to "LINE". */
token = strtok_r(line, search, &state);

/* Token will point to "TO". */
token = strtok_r(NULL, search, &state);
```


```c
#include <string.h>

char *token;
char *line = "LINE TO BE SEPARATED";
char *search = " ";
char *state;

/* Token will point to "LINE". */
token = strtok_r(line, search, &state);

/* Token will point to "TO". */
token = strtok_r(NULL, search, &state);
```

System & library functions that are not required to be thread-safe

asctime	dirname	getenv	getpwent	lgamma	readdir
basename	dlerror	getgrent	getpwnam	lgammaf	setenv
catgets	drand48	getgrgid	getpwuid	lgammal	setgrent
crypt	ecvt	getgrnam	getservbyname	localeconv	setkey
ctime	encrypt	gethostbyaddr	getservbyport	localtime	setpwent
dbm_clearerr	endgrent	gethostbyname	getservent	lrand48	setutxent
dbm_close	endpwent	gethostent	getutxent	mrand48	strerror
dbm_delete	endutxent	getlogin	getutxid	nftw	strtok
dbm_error	fcvt	getnetbyaddr	getutxline	nl_langinfo	ttyname
dbm_fetch	ftw	getnetbyname	gmtime	ptsname	unsetenv
dbm_firstkey	gcvt	getnetent	hcreate	putc_unlocked	wcstombs
dbm_nextkey	getc_unlocked	getopt	hdestroy	putchar_unlocked	wctomb
dbm_open	getchar_unlocked	getprotobynumber	inet_ntoa	pututxline	
dbm_store	getdate	getprotoent	l64a	rand	

Things to think about …

- Who gets to go next when a thread blocks/yields?
  - Scheduling!
- What happens when multiple threads are sharing the same resource?
  - Synchronization!