PROCESSES

- The thing which represents our work to the system.
- Sometimes referred to as a heavyweight process.

An instance of a program in execution.

An instance...

- May be more than one version of the same program running at the same time.
  (Hopefully sharing the code.)
  Each instance has resource limitations, security information - rights, capabilities etc.

...of a program ...

- So it includes code, data, connections (to files, networks, other processes), access to devices.

... in execution.

- It needs the processor to run. But it doesn't run all the time.
  So it needs information about what it is up to stored somewhere.

Two parts to a process

1. Resources, the things the process owns (may be shared). Also information about the process.
2. What the process is doing - the streams of execution.

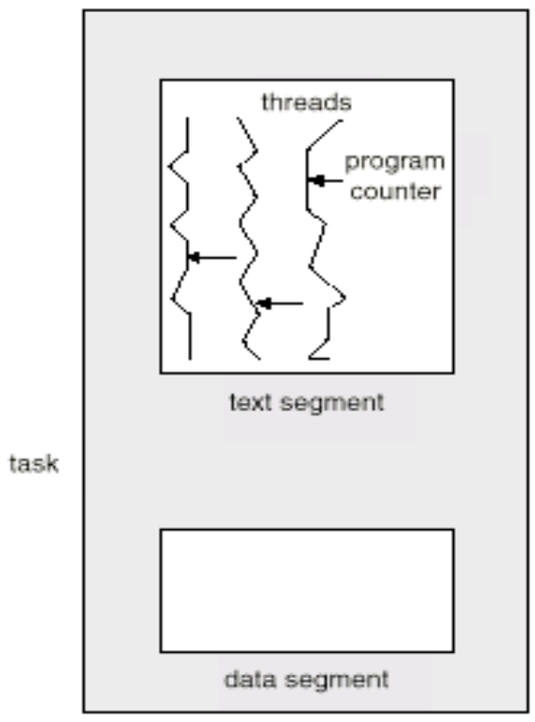
- Traditional processes had *resources* and a *single current location*. e.g. traditional UNIX.
  The resource part is called a *task* or a *job*. The location part is commonly called a *thread*.
- Most operating systems now provide support to keep these parts separate, e.g. Linux, Solaris, Windows, macOS.

Threads

Merriam-Webster definition of thread:

something continuous or drawn out: as

a : a line of reasoning or train of thought that connects the parts in a sequence (as of ideas or events) <lost the *thread* of the story>



- Sometimes referred to as lightweight processes.

A sequence of instructions being executed when there is no external intervention.

- Sometimes we want to share data as well as code. (Could just share files or memory and not use threads.)

- Easier to create than a process.
  They provide a nice encapsulation of a problem within a process rather than multiple processes.

- Easier to switch between threads than between processes.

Typical uses

- splitting work across processors (shared memory multiprocessor, multiple cores)
- added responsiveness (handle user input while still finishing another function)
- controlling and monitoring other threads
- server applications
- can help program abstraction

Thread implementation

User-level (or green threads)

- The OS only sees one thread per process.
- The process constructs other threads by user-level library calls or by hand.
- User-level control over starting and stopping threads.
- Usually a request is made to the OS to interrupt the process regularly (an alarm clock) so that the process can schedule another thread.
- The state of threads in the library code does not correspond to the state of the process.

System-level

- The OS knows about multiple threads per process.
- Threads are constructed and controlled by system calls.
- The system knows the state of each thread.

pthread creation

- You may need to hold on to the pthread information

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
        void *(*start_routine)(void *), void *arg)
```

Creates a new thread of execution.

- so you can check when it has finished

```
int pthread_join(pthread_t thread, void **value_ptr)
```

Causes the calling thread to wait for the termination of the specified thread.

```c
// simple pthread example
#include <stdio.h>
#include <pthread.h>

int counter = 0;

void *increment_counter(void *id) {
    for (int i = 0; i < 1000000; i++) {
        counter++;
    }
    printf("thread: %ld counter: %d\n", (long)id, counter);
    return (NULL);
}

int main() {
    int const num_threads = 10;
    pthread_t thread_refs[num_threads];

    for (long i = 0; i < num_threads; i++) {
        pthread_create(&thread_refs[i], NULL, increment_counter, (void *)i);
    }
    for (int i = 0; i < num_threads; i++) {
        pthread_join(thread_refs[i], NULL);
    }
}
```

User-level thread advantages

- Works even if the OS doesn't support threads.
- Some implementations of Java have user-level threads because the underlying OS doesn’t.
- Easier to create - no system call.
- Just a normal library procedure call.
- No switch into kernel mode (this saves time).
- Control can be application specific.
- Sometimes the OS doesn’t give the type of control an application needs.
- e.g. precise priority levels, changing scheduling decisions according to state changes
- Easier to switch between - saves two processor mode changes.
- Can be as simple as saving and loading registers (including SP, PSW and PC).
- So why would anyone want to use system-level threads?

Which can't be done with user-level threads?

- splitting work across processors (shared memory multiprocessor, multiple cores)
- added responsiveness (handle user input while still finishing another function)
- controlling and monitoring other threads
- server applications
- can help program abstraction

System-level thread advantages

Each thread can be treated separately.

- Rather than using the timeslice of one process over many threads.
- Should a process with 100 threads get 100 times the CPU time of a process with 1 thread?

A thread blocking in the kernel doesn't stop all other threads in the same process.

- With the user-level threads if one thread blocks for IO the OS sees the process as blocked for IO.

On a multiprocessor (including multi-core) different threads can be scheduled on different processors.

- This can only be done if the OS knows about the threads.
- Even then it sometimes doesn’t work - standard Python has system level threads but the Global Interpreter Lock (GIL) means that only one runs at a time even on a multicore machine

Jacketing

One major problem with user-level threads is the blocking of all threads within a process when one blocks.

A possible solution is known as jacketing.

- A blocking system call has a user-level jacket.
- The jacket checks to see if the resource is available, e.g., device is free.
- If not another thread is started.
- When the calling thread is scheduled again (by the thread library) it once again checks the state of the device.

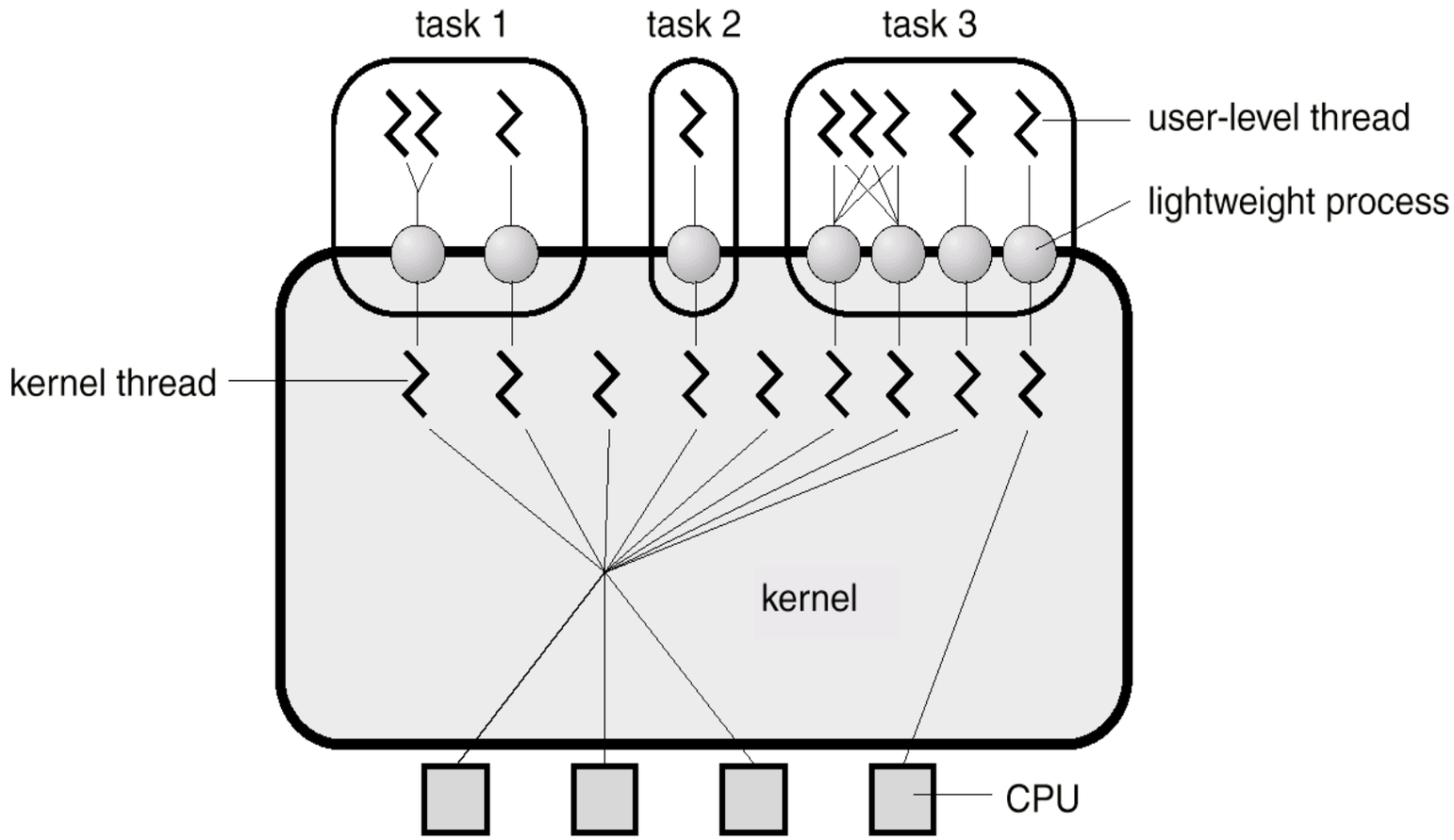
So there has to be some way of determining if resources are available to accept requests immediately.

The best of both worlds?

- Solaris (versions < 9) had both user-level and system-level threads.
- LWP – light-weight process (what we have been calling system-level threads)
- Kernel threads – active within the kernel
  - Each LWP is associated with one kernel thread.
- One or more user threads could be multiplexed on each LWP.
- A process could have several user and several LWPs.
- The number of LWPs per process was adjusted automatically to keep threads running.

From Windows 7 on there is something analogous to this with User-Mode Scheduling. https://channel9.msdn.com/Shows/Going+Deep/Dave-Probert-Inside-Windows-7-User-Mode-Scheduler-UMS

Solaris < 9 process thread system



From version 9 onwards, Solaris uses one-to-one mapping of user-level and kernel-level threads.

Original Linux threads (before 2.6)

- Clone - makes a new process (more on that later)
  Shares - memory, open files (actually descriptors), signal handlers
- From one point of view original Linux threads are processes - but they share all resources and hence the advantages of threads.

Original Linux threads and POSIX

- Can't be set to schedule threads according to priority within a process - each thread is scheduled independently across all threads/ processes in the system.
- Can't send a signal to the whole process.
- Ordinary system calls e.g. read, are not cancellation points.
- Starting a new program in one thread doesn't kill the other threads in the same process.
- When an original Linux thread blocks doing IO do all other threads in the same process stop?

Before the next lecture

Read textbook

- 3.3 Operations on Processes
- 20.4 Process Management