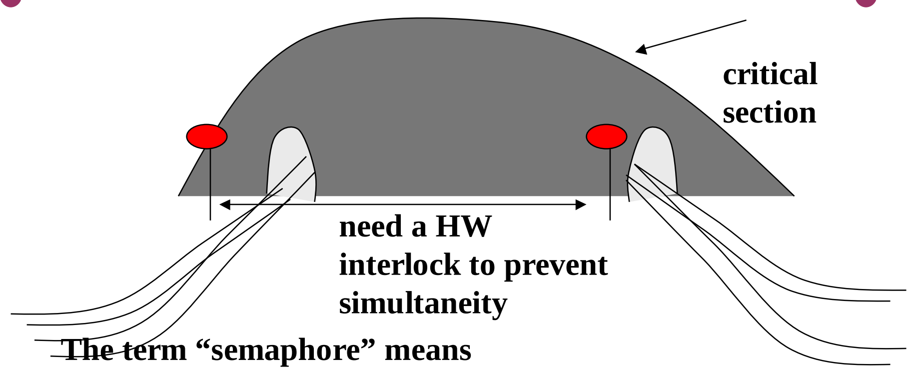
Process Synchronization and Communication

- How to we protect a critical section without disabling interrupts?

Process Synchronization



The term "semaphore" means signal. It comes from the railroads.

A semaphore requires some form of mutual exclusion in hardware: like disabling interrupts. By making it an OS call, we leave implementation up to the OS/HW. Same system call on many HW platforms.

Semaphore Implementations

- Some processors have a Test-and-Set Instruction
  - provides hardware level atomicity for synchronizing critical sections, just like using the memory mapped I/O system provides atomicity for device I/O (single instructions are atomic)
  - example:

```
bit flag
…
while (flag != false);
flag = true;
<excute code in critical section>
…
```

bad time for an interrupt?

- If processor has a test and set operation, it could look like this in ass'y code

```
loop: tst      flag, loop;        //sometimes they just skip the next instruction
< execute critical section >
```

- But we still don't want to rely on our compiler…so we use a system call

```
semaphore s;   // declare a semaphore
while(!os_set(s));   // os_set(semaphore) is a system call…OS guarantees atomicity
<execute critical section>
```


A Better Semaphore

- Problem with first semaphore: busy-waiting. OS can't tell the difference between busy waiting and doing real work?
- How can we solve this? Try a blocking semaphore

```
struct sem {
    processQueue Q;
    int     count;
};
void os_init_sem(sem *s) {
    s    count = MAX_PROC_IN_SECTION; // probably one
}
void os_wait(sem *s) {
    disable();
    s    count--;
    if (s    count < 0) {
            block calling process and put it in s    queue;
            start any process in the ready-to-run queue;
    }
    enable();
}
```

Is there and opportunity for deadlock detection?

```
sem *cs1;
….
os_wait(cs1);
<execute critical section>
os_signal(cs1);
// tiny has wait and signal
// but they are thread
// specific
```

```
void os_signal(sem *s) {
  disable();
  s    count++;
  if (s    count < 0) {
     move proc. from
          s->queue to
          ready queue;
  }
  enable();
}
```

Better than Semaphores

- They're kinda like goto…makes your multithreaded system into spaghetti
- Hard to tell if you have created a deadlock situation
- An alternative? The Monitor: you see this in Java with "synchronized" methods
  - Only one thread can be executing in a monitor at a time. The compiler builds in all of the os_calls and semaphores needed to protect the critical section
  - No chance of forgetting to signal when leaving! unless of course a process dies in the monitor!
  - Basic idea…bury semaphores in the compiler

```
class queue {
    Vector data;
    queue() { data = new Vector();}
    synchronized void put (Object x) {
        data.add(x);
    }
    synchronized Object get(Object x) {
        data.remove(0);
    }
}
```

```
class top {
    q = new queue();
    Producer p = new Producer(q);
    Consumer c = new Consumer(q);
    p.run();
    c.run();
}
```

individual methods can also be synchronized

Message Passing

- Use queues, or queues of buffers
- Use monitors/semaphores to protect the queues
- Doesn't work if we are dealing with processes rather than threads…separate address space. Process a can't just send process b a pointer! They can't access the same queue!
- So instead, provide inter-process message passing system calls
  - os_send(destination, message);
  - os_receive(source, message);
  - Maybe implemented as a critical section in the OS
- Design Consideration
  - Efficiency: do we have to copy from address space to another
  - Space: how much space is there for messages? Who allocates the space?
  - Authentication
  - End-to-End guarantees, protocol.
- We will see more of this in Linux, along with semaphores and other synchronization and communication primitives
- This also leads into networking…what if the processes are on different machines?

Back to Comparative Real Time OSes

	RTX51 Full	RTX51 Tiny
Maximum Number of Tasks	256	16
Maximum Active Tasks	19	16
CODE Space Required	6-8 Kbytes	900 Bytes
DATA Space Required	40-46 Bytes	7 Bytes
Stack (**IDATA**) Space Required	20-200 Bytes	3 Bytes for each task
XDATA Space Required	650 Bytes minimum	–
Timer Used	0, 1, or 2	0
System Clock Divisor	1,000-40,000 cycles	1,000-65,535 cycles
Interrupt Latency	< 50 cycles	< 20 cycles
Context Switch Time (Fast Task) (depends on stack load)	70-100 cycles	–
Context Switch Time (Standard Task) (depends on stack load)	180-700 cycles	100-700 cycles
Task Priority Levels	4	–
Semaphores	8 maximum	–
Mailboxes	8 maximum	–
Mailbox Size	8 entries	–
Memory Pools	16 maximum	–

Compare to uClinux at ~400Kbytes.

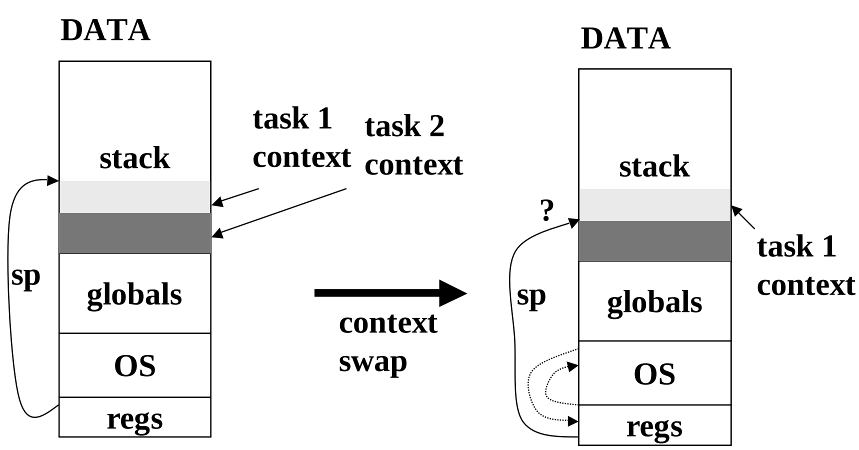
What is this?

38uS – 280uS

actually 16 semaphores

Threads and Stacks

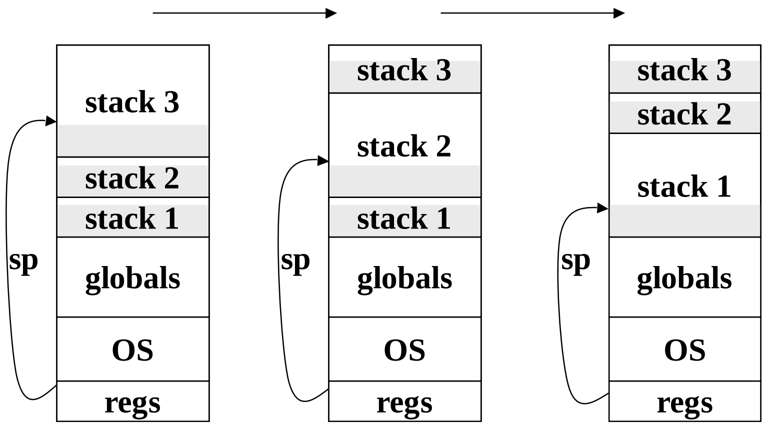
- Process: entire address space is private (processes can be multi-threaded)
- Threads: heap is public, but stack is private. What if stack wasn't private?



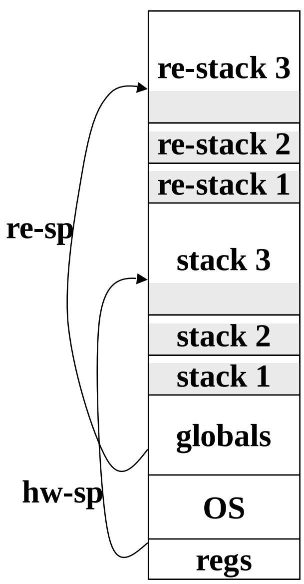
How TINY manages the stack

- One stack/thread

worst case stack size is sum of worst case for each task, plus ISR's.



Reentrant Stack



Why not use the regular (Hardware Stack) for reentrant stack frames?

Lab 4

Prove It!

- Why does Tiny limit the tasks to Reg Bank 0?
- Write a program using Tiny OS that causes a context switch to occur with a fair amount of data (calls) on the stack (least two threads). Then use the simulator to trigger an interrupt.
    - determine the latency to the start of the interrupt routine
    - Are interrupts disabled for the entire context switch process? if so then explain the experiment you did to prove this…otherwise
    - How is it possible to allow interrupts during context switch? Make sure your interrupt routine also has some subroutine calls so that you can see what happens to the stack.
- If one thread signals another thread, then executes a wait before the end of the OS timeslice, does the signalled thread start running immediately or only at the beginning of the next timeslice?
- Turn in answers to the questions along with your test code. Include clear explanations for how you arrived at your answers.
- The Keil debugger has many utilities to help you: Dissassembler, memory window, timer window for timer 0, elapsed time in machine cycles and seconds, etc.
- Turn in clearly explained incontravertible proof of your answers.

Embedded System Types

- Data flow dominated Systems
    - our music player
    - a network router
    - queues, messages, packets, routing
- Control Dominated Systems
    - Software State machines, distributed state
    - management/synchronization, e.g. power plant, autopilot, etc.