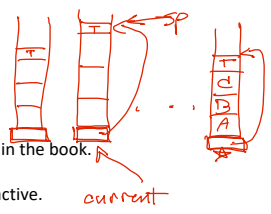
==========
Schedulers

Give us the ability to "put a thread/process to sleep" and run something else on its kernel thread/processor.

Start with coroutines
make uniprocessor run-until-block threads
add preemption
add multiple processors



Coroutines

As in Simula and Modula-2. Covered in section 8.6 in the book.

Multiple execution contexts, only one of which is active.

```
transfer(other):
   save all callee-saves registers on stack    // including ra & fp
   *current := sp
   current := other
   sp := *current
   pop all callee-saves registers     // including ra, but not sp
   return                             // into different coroutine!
```

`other` and `current` are pointers to ***context blocks***.
Each contains `sp`; may contain other stuff as well
(priority, I/O status, accounting info, etc.)

No need to change `pc`; always changes at the same place.

Create new coroutine in a state that looks like it's blocked in `transfer`.
(Or maybe let it execute and then "detach". That's basically early reply.)

Run-until block threads on a single process
Need to get rid of explicit argument to `transfer`.

`ready_list` data structure: threads that are runnable but not running.

```
reschedule:
   t : cb := dequeue(ready_list)
   transfer(t)
```

To do this safely, we need to save `current` somewhere. Two options.

(1) Suppose we're just relinquishing the processor for the sake of fairness (as in MacOS 9 or Windows 3.1):

```
yield:
   enqueue(ready_list, current)
   reschedule
```

(2) Now suppose we're implementing synchronization:

```
sleep_on(q):
   enqueue(q, current)
   reschedule
```

Some other thread/process will move us to the ready list when we can continue.

Preemption

Use timer interrupts (in OS) or signals (in library package) to trigger involuntary yields.

Requires that we protect the scheduler data structures:

```
yield:
   disable_signals()
   enqueue(ready_list, current)
   reschedule
   re-enable_signals()
```

Note that `reschedule` takes us to a different thread, possibly in code other than yield. Invariant: ***every call*** to `reschedule` must be made with signals disabled, and must re-enable them upon its return.

```
disable_signals()
if not <desired condition>
   sleep_on <condition queue>
re-enable_signals()
```

Multiprocessors

Disabling signals doesn't suffice:

```
yield:
   disable_signals()
   acquire(scheduler_lock)          // spin lock
   enqueue(ready_list, current)
   reschedule
   release(scheduler_lock)
   re-enable_signals()

disable_signals()
acquire(scheduler_lock)             // spin lock
if not <desired condition>
   sleep_on <condition queue>
release(scheduler_lock)
re-enable_signals()
```

Scheduler-Based Synchronization

semaphores

So-called ***binary*** semaphores are scheduler-based mutual exclusion locks. The acquire operation is named P; the release operation is named V; these stand for words in Dutch. (Mnemonically, I think of P as standing for "pause", though it doesn't.) Binary semaphores are called "binary" because we can think of them as a counter that is always 0 or 1, and that indicates the number of threads that could perform acquire operations without blocking.

We can extend this to ***general*** semaphores, with non-binary counters. These are useful for certain algorithms, though they don't add any additional power (you can implement them trivially with binary semaphores).

In either case, a semaphore is a special sort of counter. It has an initial value, and it keeps track of the excess (if any) of past V operations over past P operations. A P operation is delayed (the thread is de-scheduled) until #P−#V ≤ C, the initial value of the semaphore.

Here is one possible implementation:

```
type semaphore = record
   N : integer            // initialized to something non-negative
   Q : queue of threads

procedure P(ref S : semaphore) :
   disable_signals()
   acquire(scheduler_lock)
   if S.N > 0
      S.N -:= 1
   else
      sleep_on(S.Q)
   release(scheduler_lock)
   re-enable_signals()

procedure V(ref S : semaphore) :
   disable_signals()
   acquire(scheduler_lock)
   if S.Q is nonempty
      enqueue(ready_list, dequeue(S.Q))
   else
      S.N +:= 1
   release(scheduler_lock)
   re-enable_signals()
```

What can we do with semaphores?
Here is a bounded buffer:

```
shared buf : array [1..SIZE] of data
shared next_full, next_empty : integer := 1
shared mutex : semaphore := 1
shared empty_slots, full_slots : semaphore := SIZE, 0

procedure insert(d : data) :
   P(empty_slots)
   P(mutex)
   buf[next_empty] := d
   next_empty := next_empty mod SIZE + 1
   V(mutex)
   V(full_slots)

function remove returns data :
   P(full_slots)
   P(mutex)
   d : data := buf[next_full]
   next_full := next_full mod SIZE + 1
   V(mutex)
   V(empty_slots)
   return d
```

It is generally assumed that semaphores are fair, in the sense that threads complete P operations in the same order they start them.

Problems with semaphores:

(1) They're pretty low-level. When using them for mutual exclusion, for example (the most common usage), it's easy to forget a P or a V, especially when they don't occur in strictly matched pairs (because you do a V inside an if statement, for example, as in the use of the spin lock in the implementation of P).

(2) Their use is scattered all over the place. If you want to change how threads synchronize access to a data structure, you have to find all the places in the code where they touch that structure, which is difficult and error-prone.

These problems are addressed by ***monitors*** and other language mechanisms.