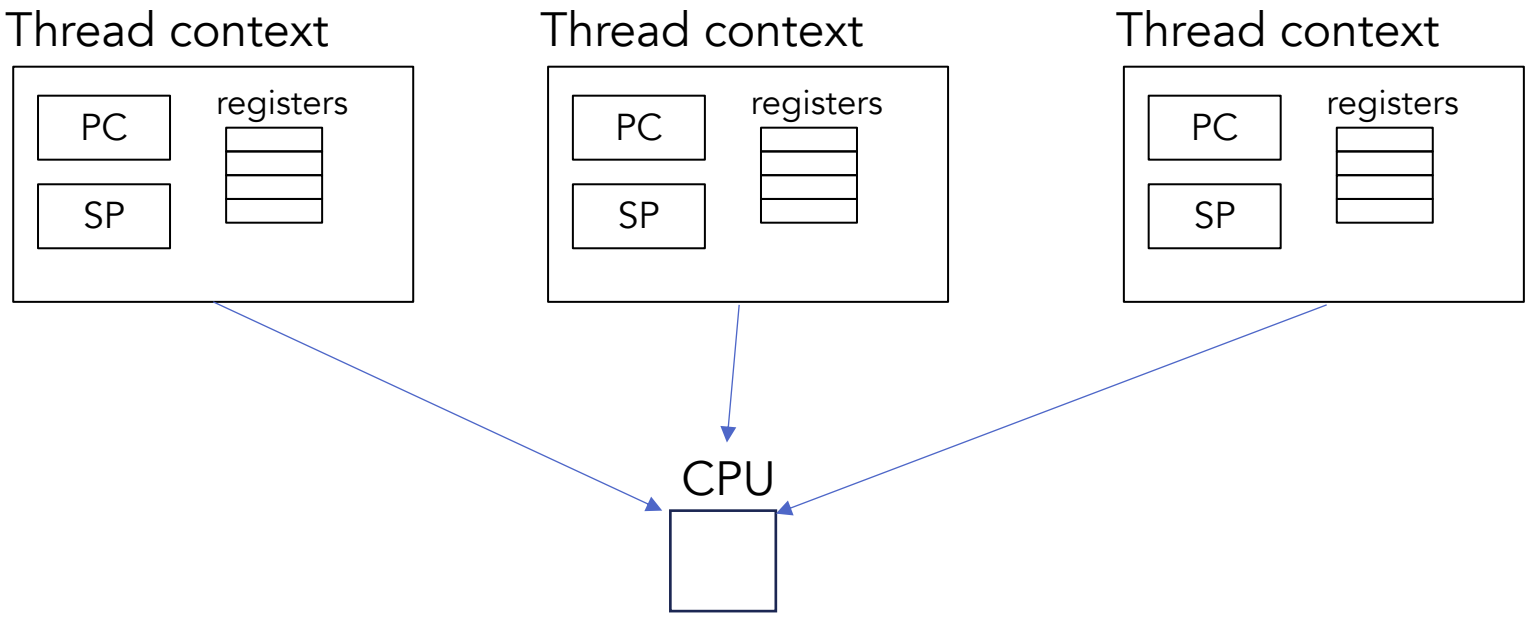
Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 7: Switching Threads

Prof. Ryan Huang

Recap: threads share the CPU



Each thread has a thread control block (TCB)

- Record execution context when the thread is not running

Recap: threads return control to OS

Internal events

- Thread calls lock(), wait(), down(), etc.
- Thread requests OS to do some work (e.g., I/O)
- Thread voluntarily gives up CPU with yield()

External events

- Initiated by something outside the thread
- Interrupts: a hardware event that transfers control from thread to OS interrupt handler
- Example: timer interrupt

Switching threads

1. Current thread returns control to OS
2. OS chooses new thread to run
3. OS saves current thread state: CPU to TCB
4. OS loads context of next thread: TCB to CPU
5. OS runs next thread

Choosing next thread to run

1 ready thread: run the ready thread

- What if there's only one thread, and that thread calls yield?

>1 ready thread: need to make a decision

- CPU's scheduling policy
- Lots of options: FIFO, priority, round robin, etc.
- Will discuss in a later lecture

0 ready threads: what should CPU do?

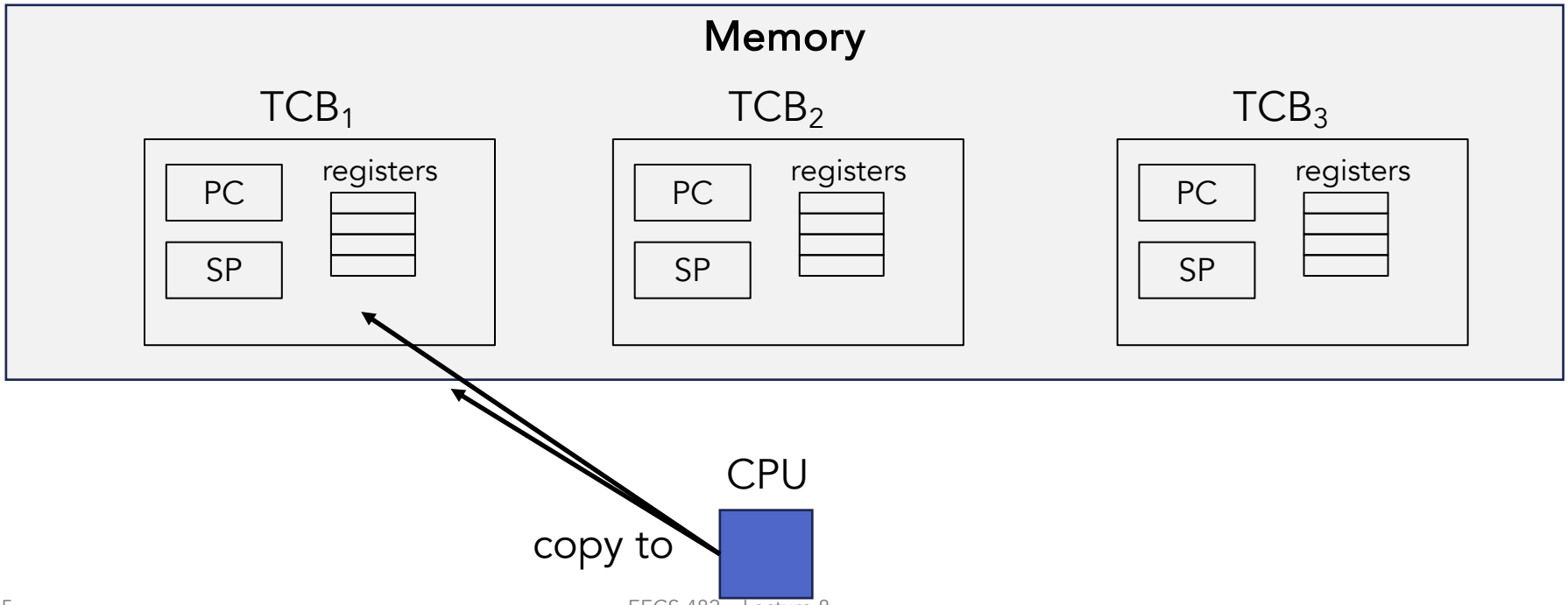
Switching threads

1. Current thread returns control to OS
2. OS chooses new thread to run
3. OS saves current thread state: CPU to TCB
4. OS loads context of next thread: TCB to CPU
5. OS runs next thread

Save state of the current thread

Save registers, stack pointer, PC

- Copy their values from CPU to memory
- Where in memory?
- Which instructions (from EECS 370)?



Save state of the current thread

Non-negligible cost

- E.g., saving floating point registers is expensive
  - optimization: only save if a thread uses floating point
- May also require switching address space (if the new thread is a different process)

Very machine dependent

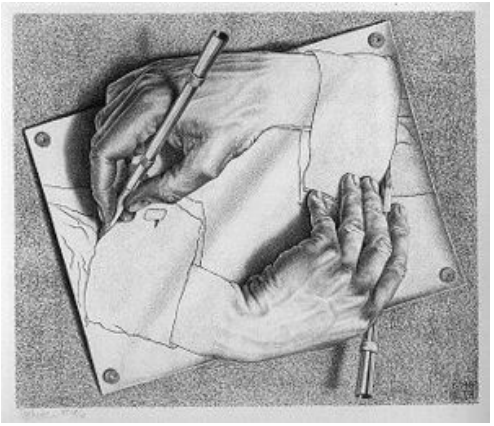
- What machine state needs to be saved and how to save them depends on the ISA

Save state of the current thread

Tricky to get right!

- Why won't the following code work?

```
100    save PC
101    switch to next thread
```



Context switch (saving + restoring) needs to be implemented in assembly language

- It requires manipulating physical registers
- It works at the level of the *calling convention* (standard for how functions should be implemented)
  - E.g., how arguments are passed, return address on the stack

Save state of the current thread

Good news: handy library functions available

- Allow user-level context switching
- Use them to implement user-level thread library (Project 2)

Glibc API: `getcontext(ucontext_t *ucp)`

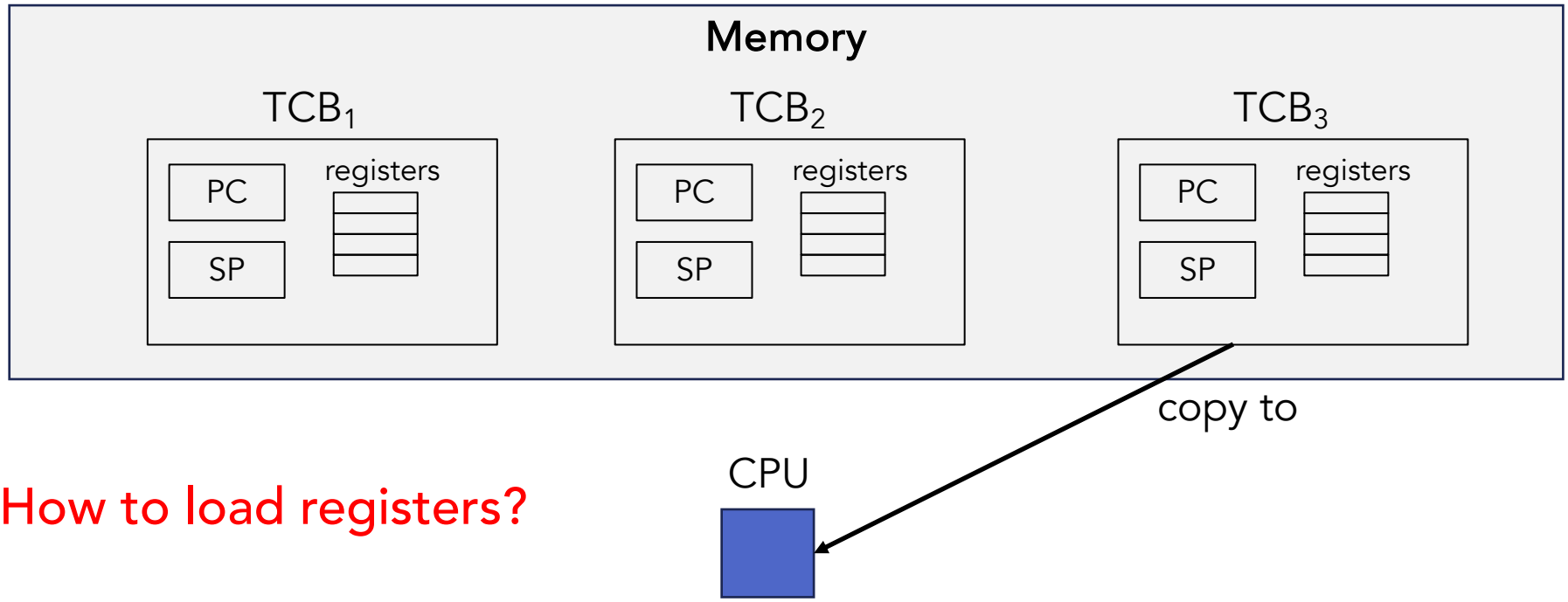
- Its implementation takes care of properly saving the current execution state into the context at `ucp`.

Switching threads

1. Current thread returns control to OS
2. OS chooses new thread to run
3. OS saves current thread state: CPU to TCB
4. OS loads context of next thread: TCB to CPU
5. OS runs next thread

Load state of the next thread

Copy from memory to CPU



How to load registers?

How to load stack pointer?

How to resume execution?

Load state of the next thread

Glibc API: `setcontext(const ucontext_t *ucp)`

- Copies the context at `ucp` from memory to CPU

How does the thread that is giving up control run again?

Switching threads

`getcontext()` copies context from CPU to memory

`setcontext()` copies context from memory to CPU

Does the following code work?

Thread A

```
10 do stuff
11 getcontext // saves state of thread A
12 setcontext to thread B
13 do more stuff
```

Thread B

```
do stuff
getcontext // saves state of thread B
setcontext to thread A
```

After B's `setcontext()`, where does A resume execution?

Where do we want A to resume execution?

Switching threads

Glibc API: `swapcontext(ucontext_t *oucp, ucontext_t *ucp)`

- Correctly combines `getcontext()` and `setcontext()`

Thread A

```
10 do stuff
11 swapcontext(A's context, B's context)
12 do more stuff
13 ...
```

Thread B

```
do stuff
swapcontext(B's context, A's context)
do more stuff
...
```

After B's `swapcontext()`, where does A resume execution?

When will B's `swapcontext()` return?

Who is carrying out these steps?

E.g., switching from thread A to B?

The CPU?

The OS?

Which thread is carrying out these steps?

Example of thread switching

Thread 1

```
print "start thread 1"
yield()
print "end thread 1"
```

Thread 2

```
print "start thread 2"
yield()
print "end thread 2"
```

Thread 1 output	Thread 2 output
start thread 1	
start yield: thread 1	
	start thread 2
	start yield: thread 2
end yield: thread 1	
end thread 1	
	end yield: thread 2
	end thread 2

yield()

```
print "start yield: thread %d"
switch to next thread (swapcontext)
print "end yield: thread %d"
```

Example of thread switching

Thread 1

```
print "start thread 1"
yield()
print "end thread 1"
```

Thread 2

```
print "start thread 2"
yield()
print "end thread 2"
```

Thread 1 output	Thread 2 output
start thread 1	
start yield: thread 1	
	start thread 2
	start yield: thread 2
end yield: thread 1	
end thread 1	
	end yield: thread 2
	end thread 2

yield()

```
print "start yield: thread %d"
switch to next thread (swapcontext)
print "end yield: thread %d"
```

Example of thread switching

Thread 1

```
print "start thread 1"
yield()
print "end thread 1"
```

Thread 2

```
print "start thread 2"
yield()
print "end thread 2"
```

Thread 1 output

```
start thread 1
start yield: thread 1
start thread 2
start yield: thread 2
end yield: thread 1
end thread 1
end yield: thread 2
end thread 2
```

yield()

```
print "start yield: thread %d"
switch to next thread (swapcontext)
print "end yield: thread %d"
```

Creating a new thread

Create a running thread? Seems challenging

Instead, create ready thread

- Key idea: make it look like it was running, put on ready queue
- Then just wait for it to be scheduled!

Construct TCB in the state it would be in if it paused at start of its initial function

How to create a thread

1. Allocate TCB

2. Allocate stack

3. Initialize context in TCB
   - New thread should look like it was about to call a function
   - Set PC to start of a function
   - Set general-purpose registers and stack to func parameters
   - Linux: `makecontext()` initializes context and stack

4. Add TCB to ready queue

Administration

You can now do most of Project 2

- cpu::cpu
- Thread create, yield, join
- Mutex lock, unlock
- CV wait, signal, broadcast

This is a very hard project *conceptually*

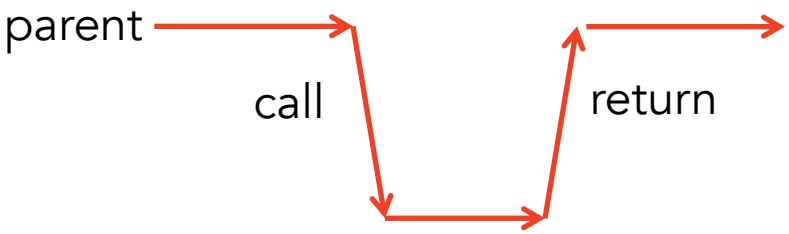
Urge you to start early!

A brief detour back to writing concurrent programs…

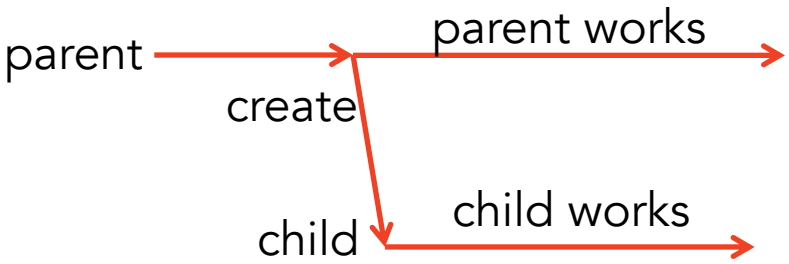
How to use new thread

Creating a thread is similar to a procedure call

Synchronous procedure call (EECS 280)

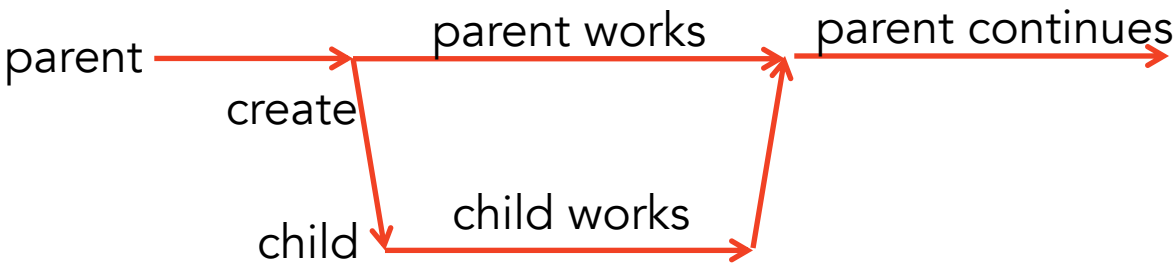


Asynchronous procedure call (EECS 482)



Synchronizing with child

What if parent wants to work for a while, then wait for child to finish?

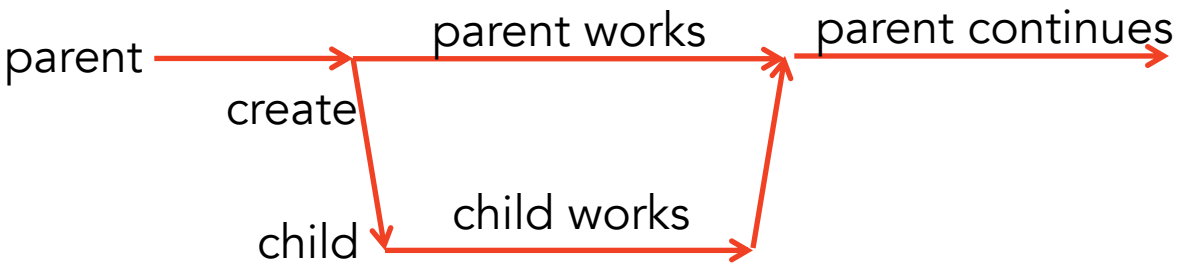


Synchronizing with child

```
parent()
    create child thread
    print "parent works"
    print "parent continues"
```

```
child()
    print "child works"
```

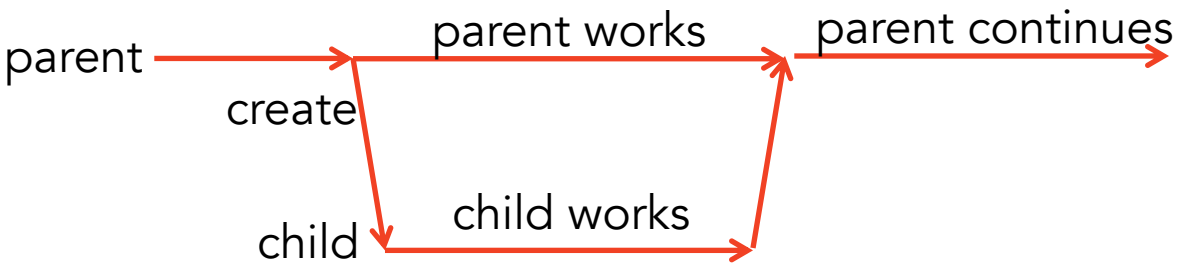
Would this work?



Synchronizing with child: yield?

```
parent()
    create child thread
    print "parent works"
    yield?
    print "parent continues"
```

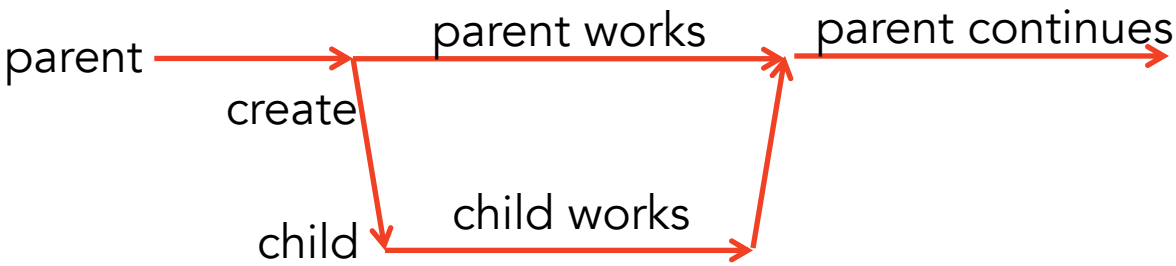
```
child()
    print "child works"
```



Synchronizing with child: monitors?

```
parent()
    create child thread
    lock
    print "parent works"
    wait?
    print "parent continues"
    unlock
```

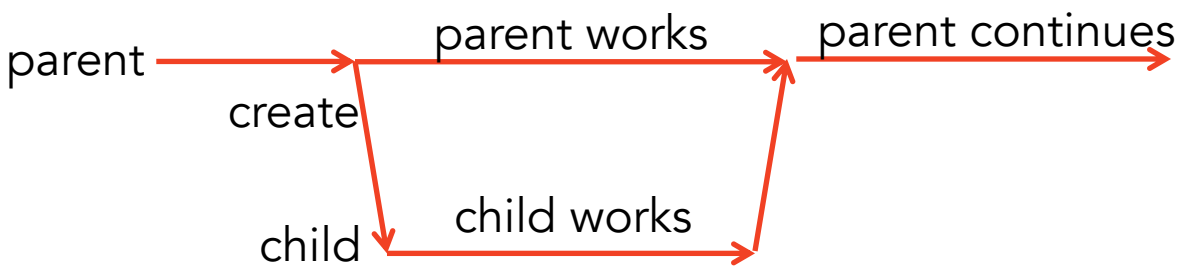
```
child()
    lock
    print "child works"
    signal?
    unlock
```



Synchronizing with child: monitors?

```
parent()
    create child thread
    lock
    print "parent works"
    while (!childDone)
      wait
    print "parent continues"
    unlock
```

```
child()
    lock
    print "child works"
    childDone = 1
    signal
    unlock
```



Synchronizing with child: join

```
parent()
    create child thread
    print "parent works"
    child.join
    print "parent continues"
```

```
child()
    print "child works"
```