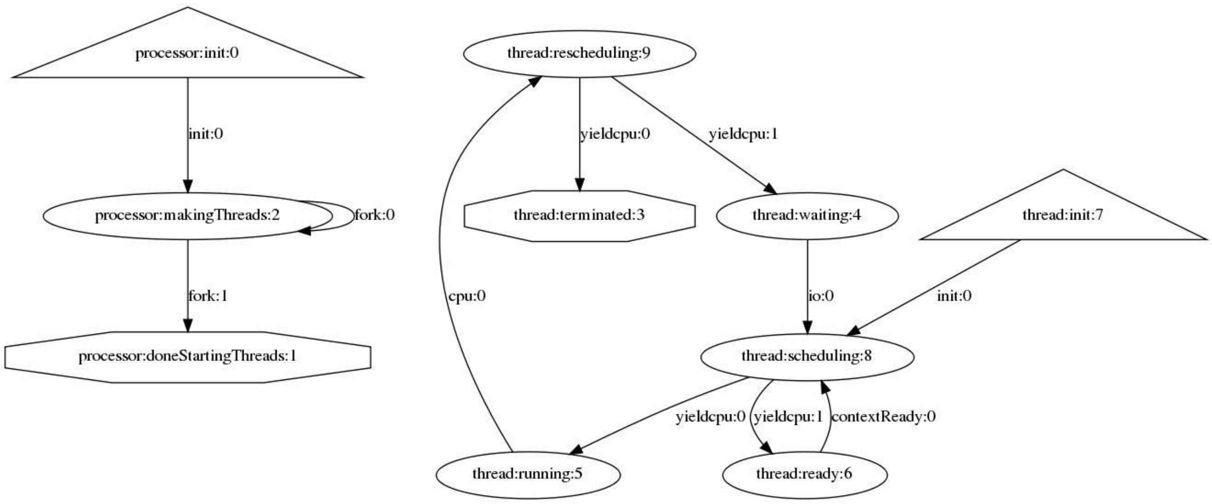
CSC 343 Operating Systems, Dr. Dale Parson, Fall 2014

State Machine Models of CPU Scheduling Algorithms

Located in ~parson/OpSys/state2codeV10

http://acad.kutztown.edu/~parson/fcfs.jpg (Substitute sjf or rr for fcfs)



fcfs.stm – First Come, First Served non-preemptive context scheduling

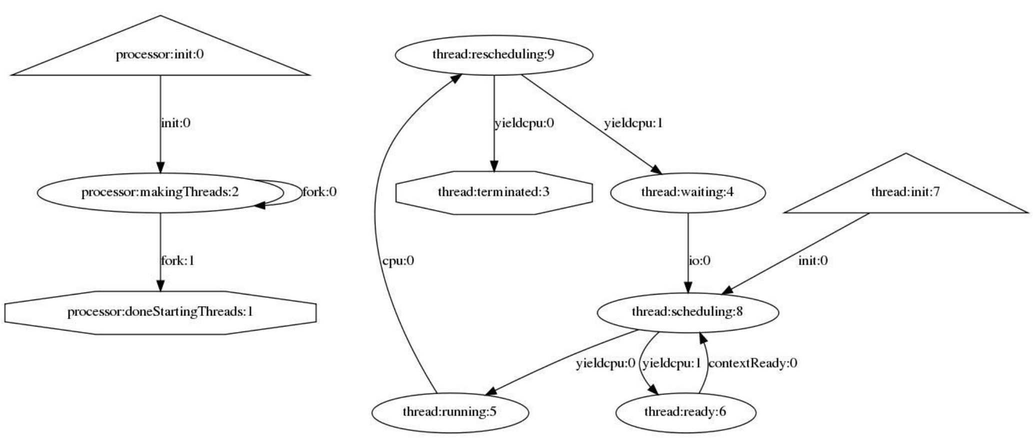
```
# CSC 343, Fall 2013, STUDENT NAME:
# fcfs.stm implements a first-come, first-served, non-preemptive scheduler as an
# example for assignment 2. D. Parson.

machine processor {
   # Use this machine in all of your files in assignment 2 to start threads.
   # It starts 10 threads, one every tick. I am starting them quickly so that
   # algorithms like FCFS don't get swamped too much an early CPU-bound thread.
   threadsToGo = 10 ;
   start init, state makingThreads, accept doneStartingThreads ;
   init -> makingThreads init()[]/@
      processor.readyq = Queue(ispriority=False);
      threadsToGo -= 1 ; fork()@
   makingThreads -> makingThreads fork(pid, tid)[@threadsToGo > 0@]/@
      threadsToGo -= 1 ; fork()@
   makingThreads -> doneStartingThreads fork(pid, tid)[@threadsToGo == 0@]/
}

# For all parts of your assignment, half of every ten threads must be IO bound.
# The others are CPU bound. The I/O bound threads must request between
# 1 and 250 ticks using the exponential sampler, with the knee of the curve
# (half of the sampled values) at 25 ticks. The CPU bound threads must
# request between 100 and 1100 ticks using the revexponential sampler,
# with the knee of the curve (half of the sampled values) at 1000 ticks.
# All threads must continue to run until simulation time is >= 100,000.
machine thread {
   machineid = -1, pid = -1, tid = -1, iobound = @False@, endtime = 100000 ;
   # Python treats 0 as False (not iobound) and 1 as True (iobound).
   # The transition out of state init initializes the above variables.
   start init, state scheduling, state ready, state running, state waiting,
```

```
        state rescheduling, accept terminated ;
    init -> scheduling init()[]/@machineid, pid, tid = getid();
        iobound = True if ((pid % 2) == 1) else False ; yieldcpu()@
        # ^^^ The odd pids are IO bound.
        # The others (50%) are CPU bound. This job mix stresses the scheduling
        # algorithms better than a strictly IO-bound or CPU-bound mix.
    scheduling -> running yieldcpu()[@processor.contextsFree > 0@]/@
        processor.contextsFree -= 1 ;
        ticks = sample(1, 250, 'exponential', 25) if iobound
            else sample(100, 1100, 'revexponential', 1000);
        msg('pid ' + str(pid) + ' tid ' + str(tid)
            + ' about to CPU for ' + str(ticks) + ' ticks');
        cpu(ticks)@
    scheduling -> ready yieldcpu()[@processor.contextsFree == 0@]/@
        # Put myself in processor's readyq with FIFO priority.
        msg('pid ' + str(pid) + ' tid ' + str(tid)
            + ' about to wait, ready for CPU');
        processor.readyq.enq(thread); waitForEvent('contextReady', False)@
    ready -> scheduling contextReady()[]/@yieldcpu()@
    running -> rescheduling cpu()[]/@
        processor.contextsFree += 1 ;
        msg('thread ' + str(tid) + ' checking readyq '
            + str(processor.readyq.len())
            + ' with contextsFree ' + str(processor.contextsFree));
        signalEvent(processor.readyq.deq(), 'contextReady')
            if len(processor.readyq) > 0 else noop();
        yieldcpu()@
    rescheduling -> terminated yieldcpu()[@time() >= endtime@]/
    rescheduling -> waiting yieldcpu()[@time() < endtime@]/@
        # iodevice of -1 (process terminal) or one of the fastio devices.
        iodevice = sample(-1, len(processor.fastio)-1, 'uniform');
        msg('thread ' + str(tid) + ' blocking on IO unit ' + str(iodevice));
        msg('pid ' + str(pid) + ' tid ' + str(tid)
            + ' about to IO on dev ' + str(iodevice));
        io(iodevice)@
    waiting -> scheduling io()[]/@yieldcpu()@
}

processor
```



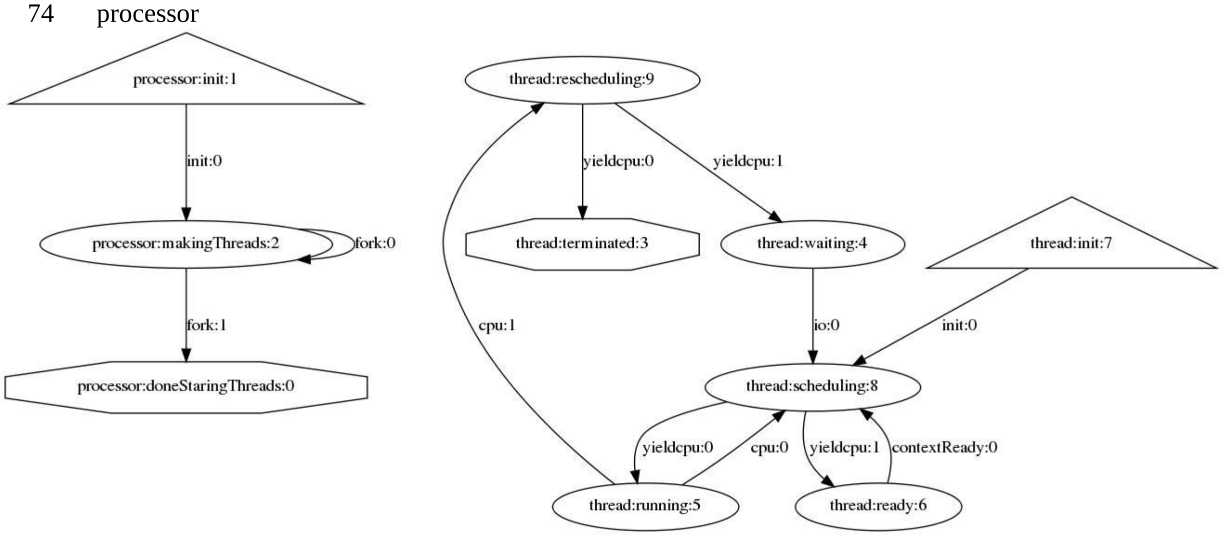
sjf.stm – non-preeemptive Shortest Job First

```
# CSC 343, Fall 2013, STUDENT NAME:
# sjf.stm implements a short-job first, non-preemptive scheduler as a
# partial solution of assignment 2. D. Parson.

machine processor {
    # Use this machine in all of your files in assignment 2 to start threads.
    # It starts 10 threads, one every tick. I am starting them quickly so that
    # algorithms like FCFS don't get swamped too much an early CPU-bound thread.
    threadsToGo = 10 ;
    start init, state makingThreads, accept doneStartingThreads ;
    init -> makingThreads init()[]/@
        processor.readyq = Queue(ispriority=True);
        threadsToGo -= 1 ; fork()@
    makingThreads -> makingThreads fork(pid, tid)[@threadsToGo > 0@]/@
        threadsToGo -= 1 ; fork()@
    makingThreads -> doneStartingThreads fork(pid, tid)[@threadsToGo == 0@]/
}

# For all parts of your assignment, half of every ten threads must be IO bound.
# The others are CPU bound. The I/O bound threads must request between
# 1 and 250 ticks using the exponential sampler, with the knee of the curve
# (half of the sampled values) at 25 ticks. The CPU bound threads must
# request between 100 and 1100 ticks using the revexponential sampler,
# with the knee of the curve (half of the sampled values) at 1000 ticks.
# All threads must continue to run until simulation time is >= 100,000.
machine thread {
    machineid = -1, pid = -1, tid = -1, iobound = @False@, endtime = 100000 ;
    # Python treats 0 as False (not iobound) and 1 as True (iobound).
    # The transition out of state init initializes the above variables.
    start init, state scheduling, state ready, state running, state waiting,
        state rescheduling, accept terminated ;
    init -> scheduling init()[]/@machineid, pid, tid = getid();
        iobound = True if ((pid % 2) == 1) else False ;
        # ^^^ The odd pids are IO bound.
        # The others (50%) are CPU bound. This job mix stresses the scheduling
        # algorithms better than a strictly IO-bound or CPU-bound mix.
        # Set ticks when going into scheduling.
        ticks = sample(1, 250, 'exponential', 25) if iobound
            else sample(100, 1100, 'revexponential', 1000);
        yieldcpu()@
    scheduling -> running yieldcpu()[@processor.contextsFree > 0@]/@
        processor.contextsFree -= 1 ;
        msg('pid ' + str(pid) + ' tid ' + str(tid) + ' about to CPU for '
            + str(ticks) + ' ticks'); cpu(ticks)@
    scheduling -> ready yieldcpu()[@processor.contextsFree == 0@]/@
        # Put myself in processor's readyq with sjf priority.
        msg('pid ' + str(pid) + ' tid ' + str(tid)
            + ' about to wait, ready for CPU ticks ' + str(ticks));
        processor.readyq.enq(thread, ticks); waitForEvent('contextReady', False)@
    ready -> scheduling contextReady()[]/@yieldcpu()@
    # ^^^ Do not set ticks coming out of ready, not used yet.
    running -> rescheduling cpu()[]/@
        processor.contextsFree += 1 ;
```

```
        msg('thread ' + str(tid) + ' checking readyq '
            + str(len(processor.readyq))
            + ' with contextsFree ' + str(processor.contextsFree));
        signalEvent(processor.readyq.deq(), 'contextReady')
            if len(processor.readyq) > 0 else noop();
        yieldcpu()@
    rescheduling -> terminated yieldcpu()[@time() >= endtime@]/
    rescheduling -> waiting yieldcpu()[@time() < endtime@]/@
        # iodevice of -1 (process terminal) or one of the fastio devices.
        iodevice = sample(-1, len(processor.fastio)-1, 'uniform');
        msg('thread ' + str(tid) + ' blocking on IO unit ' + str(iodevice));
        msg('pid ' + str(pid) + ' tid ' + str(tid) + ' about to IO on dev '
            + str(iodevice));
        io(iodevice)@
    waiting -> scheduling io()[]/@
        ticks = sample(1, 250, 'exponential', 25) if iobound
            else sample(100, 1100, 'revexponential', 1000);
        yieldcpu()@
}
processor
```



rr.stm – Round Robin Preemptive Scheduling

```
# CSC 343, Fall 2013, STUDENT NAME:
# rr.stm implements a preemptive round-robin scheduler as a
# partial solution of assignment 2. D. Parson.
machine processor {
    # Use this machine in all of your files in assignment 2 to start threads.
    # It starts 10 threads, one every tick. I am starting them quickly so that
    # algorithms like FCFS don't get swamped too much an early CPU-bound thread.
    threadsToGo = 10 ;
    start init, state makingThreads, accept doneStaringThreads ;
    init -> makingThreads init()[]/@
        processor.readyq = Queue(ispriority=False);
        threadsToGo -= 1 ; fork()@
    makingThreads -> makingThreads fork(pid, tid)[@threadsToGo > 0@]/@
        threadsToGo -= 1 ; fork()@
    makingThreads -> doneStaringThreads fork(pid, tid)[@threadsToGo == 0@]/
}
```

```
# For all parts of your assignment, half of every ten threads must be IO bound.
# The others are CPU bound. The I/O bound threads must request between
# 1 and 250 ticks using the exponential sampler, with the knee of the curve
# (half of the sampled values) at 25 ticks. The CPU bound threads must
# request between 100 and 1100 ticks using the revexponential sampler,
# with the knee of the curve (half of the sampled values) at 1000 ticks.
# All threads must continue to run until simulation time is >= 100,000.
# STUDENT: The quantum must be 125 ticks. Make sure that a thread never
# calls cpu() with more than quantum ticks; use Python's min(a, b) function.
# Make sure to keep any un-run ticks returns from sample() in a variable,
# and make sure that as long as the remaining ticks from the most
# recent sample() have not reached 0, that your thread gets back to the
# ready state (processor.readyq.enq) WITHOUT doing io(). It should request
# io() EXACTLY at the point that it has consumed all ticks supplied
# by the most recent sample() call, after which it can sample() a new
# CPU-burst number of ticks. Any given cpu() call must NEVER exceed
# the quantum limit.
machine thread {
    quantum = 125, machineid = -1, pid = -1, tid = -1, iobound = @False@,
        endtime = 100000 ;
    # Python treats 0 as False (not iobound) and 1 as True (iobound).
    # The transition out of state init initializes the above variables.
    start init, state scheduling, state ready, state running, state waiting,
        state rescheduling, accept terminated ;
    init -> scheduling init()[]/@machineid, pid, tid = getid();
        iobound = True if ((pid % 2) == 1) else False ;
        # ^^^ The odd pids are IO bound.
        # The others (50%) are CPU bound. This job mix stresses the scheduling
        # algorithms better than a strictly IO-bound or CPU-bound mix.
        # Set ticks when going into scheduling.
        ticks = sample(1, 250, 'exponential', 25) if iobound
            else sample(100, 1100, 'revexponential', 1000);
        tickstorun = min(ticks, quantum);
        tickstodefer = ticks - tickstorun;
        yieldcpu()@
        # ^^^ pids that give a remainder of 5 for divide-by-10 are CPU bound.
    scheduling -> running yieldcpu()[@processor.contextsFree > 0@]/@
        processor.contextsFree -= 1 ;
        msg('pid ' + str(pid) + ' tid ' + str(tid) + ' about to CPU for '
            + str(tickstorun) + ' tickstorun ' + ' out of ' + str(ticks)
            + ' ticks, tickstodefer = ' + str(tickstodefer));
        cpu(tickstorun)@
    scheduling -> ready yieldcpu()[@processor.contextsFree == 0@]/@
        # Put myself in processor's readyq with rr priority.
        msg('pid ' + str(pid) + ' tid ' + str(tid)
            + ' about to wait, ready for CPU tickstorun ' + str(tickstorun)
            + ' out of ' + str(ticks) + ' ticks, tickstodefer = '
            + str(tickstodefer));
        processor.readyq.enq(thread); waitForEvent('contextReady', False)@
    ready -> scheduling contextReady()[]/@yieldcpu()@
    # ^^^ Do not set ticks; they have not all been used.
    running -> scheduling cpu()[@tickstodefer > 0@]/@
        processor.contextsFree += 1 ;
        signalEvent(processor.readyq.deq(), 'contextReady')
            if len(processor.readyq) > 0 else noop();
        tickstorun = min(tickstodefer, quantum);
```

```
        tickstodefer = tickstodefer - tickstorun;
        yieldcpu()@
    running -> rescheduling cpu()[@tickstodefer  < 1@]/@
        processor.contextsFree += 1 ;
        signalEvent(processor.readyq.deq(), 'contextReady')
            if len(processor.readyq) > 0 else noop();
        yieldcpu()@
    rescheduling -> terminated yieldcpu()[@time() >= endtime@]/
    rescheduling -> waiting yieldcpu()[]/@
        # Pick an iodevice of -1 (process terminal) or one of the fastio devices.
        iodevice = sample(-1, len(processor.fastio)-1, 'uniform');
        msg('thread ' + str(tid) + ' blocking on IO unit ' + str(iodevice));
        msg('pid ' + str(pid) + ' tid ' + str(tid) + ' about to IO on dev '
            + str(iodevice));
        io(iodevice)@
    waiting -> scheduling io()[]/@
        ticks = sample(1, 250, 'exponential', 25) if iobound
            else sample(100, 1100, 'revexponential', 1000);
        tickstorun = min(ticks, quantum);
        tickstodefer = ticks - tickstorun;
        yieldcpu()@
}

processor
```