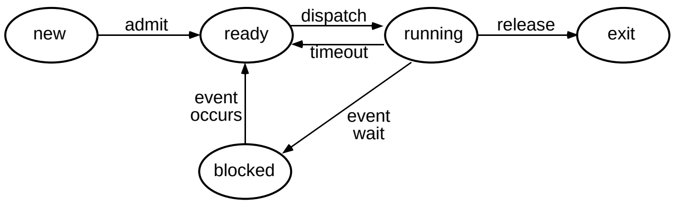
CPU Scheduling



- The *CPU scheduler* (sometimes called the *dispatcher* or *short-term scheduler*):
  - Selects a process from the ready queue and lets it run on the CPU
    - Assumes all processes are in memory, and one of those is executing on the CPU
  - Crucial in multiprogramming environment
    - Goal is to maximize CPU utilization
- *Non-preemptive* scheduling — scheduler executes only when:
  - Process is terminated
  - Process switches from running to blocked

Process Execution Behavior

- Assumptions:
  - One process per user
  - One thread per process
  - Processes are independent, and compete for resources (including the CPU)
- Processes run in CPU - I/O burst cycle:
  - Compute for a while (on CPU)
  - Do some I/O
  - Continue these two repeatedly
- Two types of processes:
  - CPU-bound — does mostly computation (long CPU burst), and very little I/O
  - I/O-bound — does mostly I/O, and very little computation (short CPU burst)

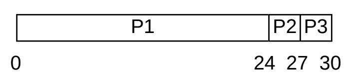
First-Come-First-Served (FCFS)

- Other names:
  - First-In-First-Out (FIFO)
  - Run-Until-Done
- Policy:
  - Choose process from ready queue in the order of its arrival, and run that process non-preemptively
    - Early FCFS schedulers were overly non-preemptive: the process did not relinquish the CPU until it was finished, even when it was doing I/O
    - Now, non-preemptive means the scheduler chooses another process when the first one terminates or blocks
- Implement using FIFO queue (add to tail, take from head)
- Used in Nachos (as distributed)

FCFS Example

- Example 1:

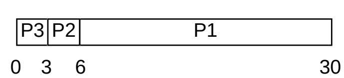
Process (Arrival Order)	**P1**	**P2**	**P3**
Burst Time	24	3	3
Arrival Time	0	0	0



average waiting time = (0 + 24 + 27) / 3 = 17

- Example 2:

Process (Arrival Order)	**P3**	**P2**	**P1**
Burst Time	3	3	24
Arrival Time	0	0	0



average waiting time = (0 + 3 + 6) / 3 = 3

Scheduling in Nachos (Review)

- main( ) (in threads/main.cc)
  calls Initialize( ) (in threads/system.cc)
  - which starts scheduler, an instance of class Scheduler (defined in **threads/scheduler.h**, **scheduler.cc**)
- Interesting functions:
  - Mechanics of running a thread:
    - Scheduler::ReadyToRun( ) — puts a thread at the tail of the ready queue
    - Scheduler::FindNextToRun( ) — returns thread at the head of the ready queue
    - Scheduler::Run( ) — switches to thread
  - Scheduler is non-preemptive FCFS — chooses next process when:
    - Current thread terminates
    - Current thread calls Thread::Yield( ) to explicitly yield the CPU
    - Current thread calls Thread::Sleep( ) (to block (wait) on some event)

Scheduling in Nachos (Review)

```
Scheduler::Scheduler ( )
{
    readyList = new List;
}

void
Scheduler::ReadyToRun (Thread *thread)
{
   DEBUG('t',
      "Putting thread %s on ready list.\n",
      thread->getName());
   thread->setStatus(READY);
   readyList->Append((void *)thread);
}

Thread *
Scheduler::FindNextToRun ( )
{
    return (Thread *)readyList->Remove();
}
```


Scheduling in Nachos (Review)

```
void
Scheduler::Run (Thread *nextThread)
{
   Thread *oldThread = currentThread;

   oldThread->CheckOverflow();
   currentThread = nextThread;
   currentThread->setStatus(RUNNING);

   DEBUG('t', "Switching from thread \"%s\"
   to thread \"%s\"\n",oldThread->getName(),
      nextThread->getName());
   SWITCH(oldThread, nextThread);
   DEBUG('t', "Now in thread \"%s\"\n",
      currentThread->getName());

   if (threadToBeDestroyed != NULL) {
      delete threadToBeDestroyed;
      threadToBeDestroyed = NULL;
   }
}
```


Manipulating Threads in Nachos (Review)

```
void
Thread::Fork(VoidFunctionPtr func, int arg)
{
   DEBUG('t',"Forking thread \"%s\" with
      func = 0x%x, arg = %d\n",
      name, (int) func, arg);

   StackAllocate(func, arg);

   IntStatus oldLevel = interrupt->
      SetLevel(IntOff);
   scheduler->ReadyToRun(this);
   (void) interrupt->SetLevel(oldLevel);
}
```

Manipulating Threads in Nachos (cont.)

```
void
Thread::Yield ()
{
   Thread *nextThread;

   IntStatus oldLevel = interrupt->
      SetLevel(IntOff);

   ASSERT(this == currentThread);
   DEBUG('t', "Yielding thread \"%s\"\n",
      getName());

   nextThread = scheduler->
      FindNextToRun();
   if (nextThread != NULL) {
      scheduler->ReadyToRun(this);
      scheduler->Run(nextThread);
   }
   (void) interrupt->SetLevel(oldLevel);
}
```


Manipulating Threads in Nachos (cont.)

```
void
Thread::Sleep ()
{
   Thread *nextThread;

   ASSERT(this == currentThread);
   ASSERT(interrupt->getLevel() == IntOff);
   DEBUG('t', "Sleeping thread \"%s\"\n",
      getName());

   status = BLOCKED;
   while ((nextThread = scheduler->
      FindNextToRun()) == NULL)
      interrupt->Idle();

   scheduler->Run(nextThread);
}
```


Semaphores in Nachos (Review)

```
void
Semaphore::P()
{
   IntStatus oldLevel = interrupt->
      SetLevel(IntOff);    // disable interrupts

   while (value == 0) {        // sema not avail
      queue->                  // so go to sleep
          Append((void *)currentThread);
      currentThread->Sleep();
   }

   value--;              // semaphore available,
                         // consume its value

   (void) interrupt->    // re-enable interrupts
      SetLevel(oldLevel);
}
```


Semaphores in Nachos (cont.) (Review)

```
void
Semaphore::V()
{
   Thread *thread;

   IntStatus oldLevel = interrupt->
      SetLevel(IntOff);

   thread = (Thread *)queue->Remove();
   if (thread != NULL) // make thread ready,
             // consuming the V immediately
      scheduler->ReadyToRun(thread);

   value++;

   (void) interrupt->SetLevel(oldLevel);
}
```