Scheduling and schedulers

Dr. Bystrov

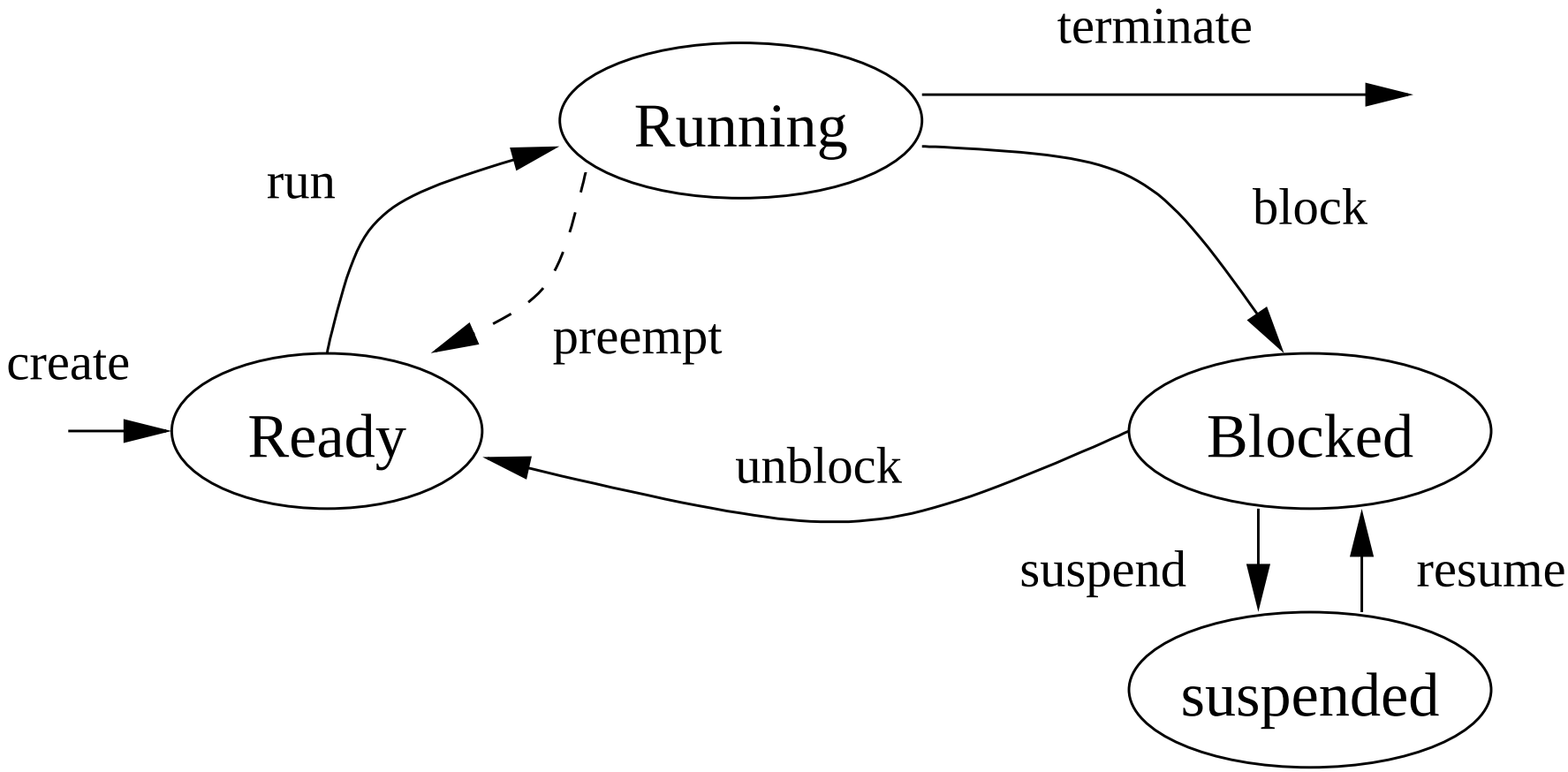
School of Electrical Electronic and Computer Engineering

University of Newcastle upon Tyne

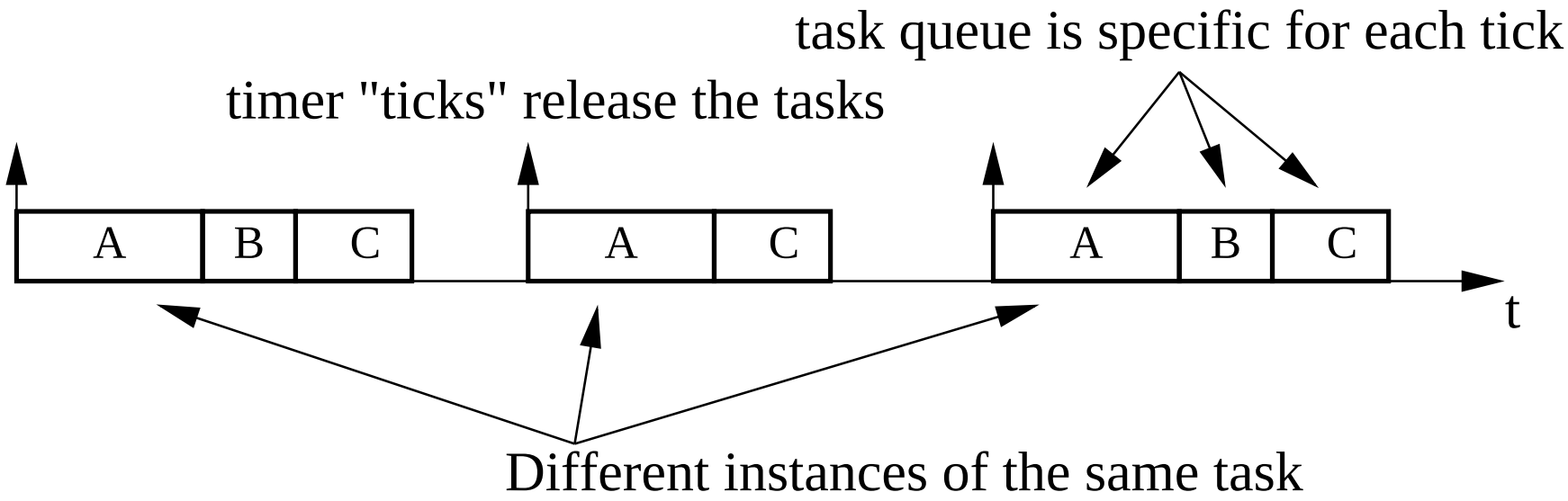
Inside a scheduler

- Interrupts
- Timer
- One or several processors
- Task queue
- CPU modes (user, supervisor, IRQ, SWI, etc.)
- Task model
- Scheduler model
- Protection from overruns, etc.

Task model (generic)

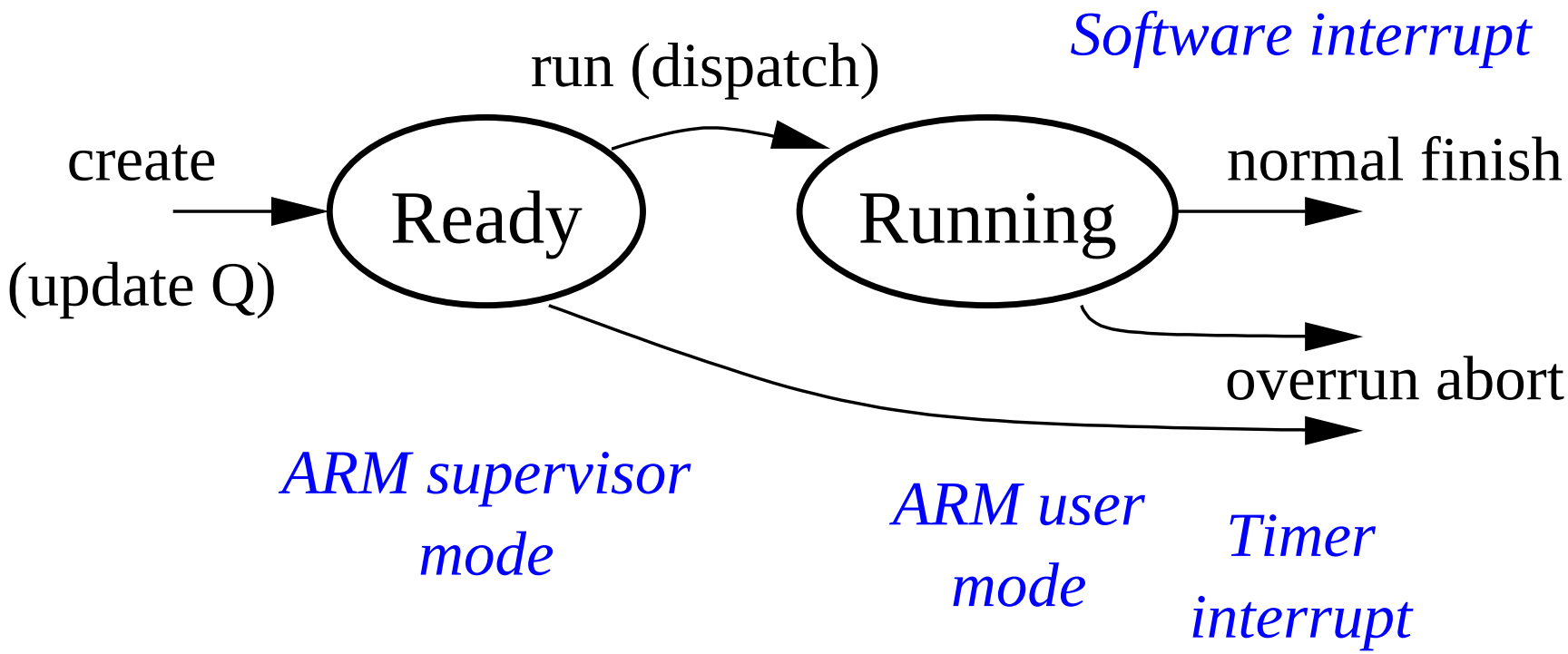


Simple Timeline TTS



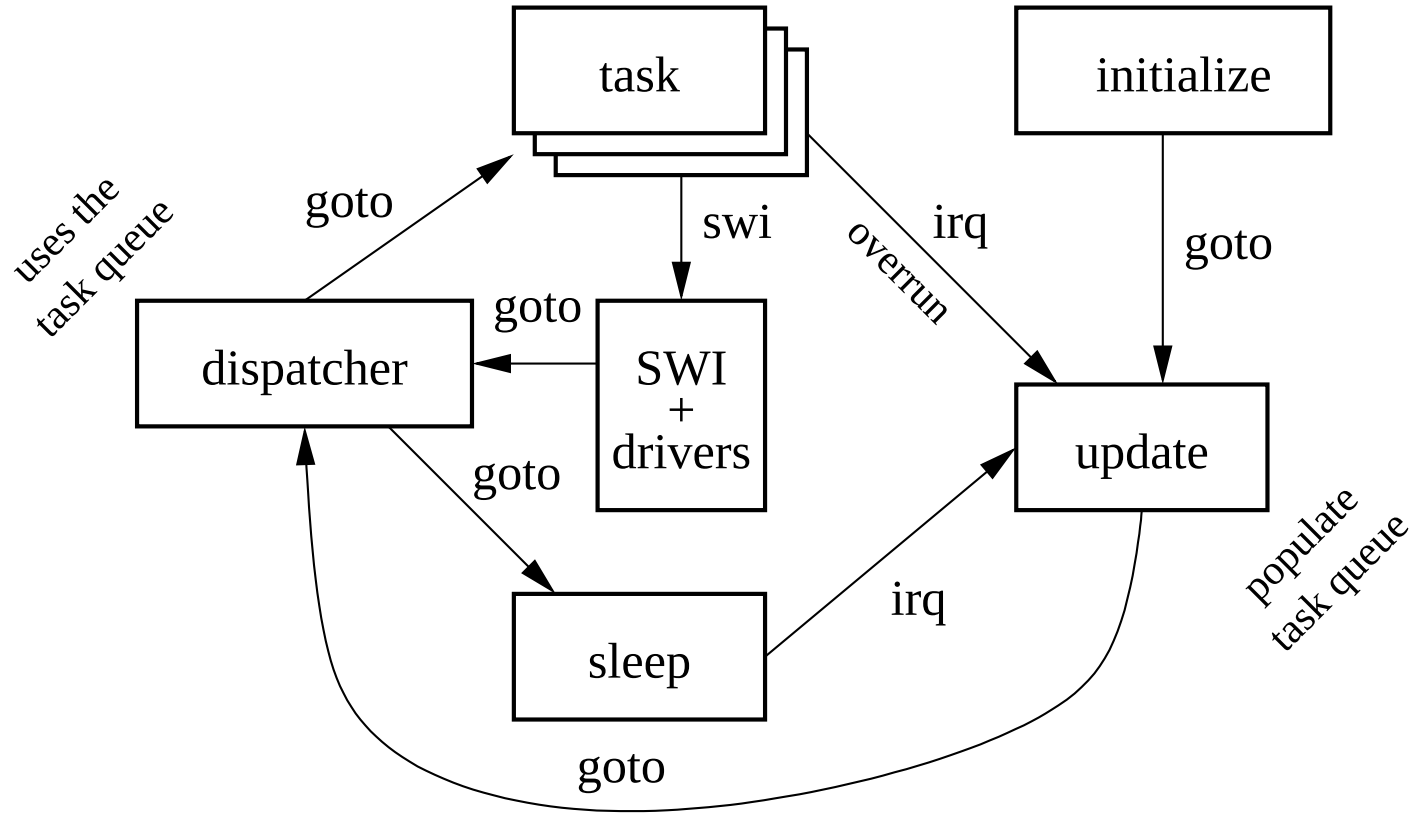
- Short periodic tasks, deadlines coincide with the next release
- Cooperative
- Static, off-line, trivially optimal
- Very simple,predictable, fast
- Overruns, only short periodic tasks

Timeline TTS task model



- Task queue is updated on each tick, which “creates” the tasks. Their code is stored in ROM.
- An overrun may happen either either when the task is running or while it still pending (in the previous task).

Timeline TTS state graph



- This is an FSM model, not a subroutine call tree!
- “Naked” functions, goto, stack init in each block.

Timeline TTS C-code

- ARM platform – supervisor (SVC), interrupt (IRQ), user (USER) modes use different registers
- “Naked” functions – no prologue/epilogue, stack needs to be reinitialised at the beginning
- No-return calls
- The only way to switch from USER to SVC is to use the software interrupt instruction
- FSM (goto) programming paradigm
- Trees of function calls only exist locally inside the blocks, e.g. inside the tasks

See the distributed example of C-code.

Short periodic task programming

- We need to implement a continuous process
- ... but only short periodic tasks are available
  - isolate the main loop
  - represent it’s body as a task
  - “stitch” the task instances together with static data

Proportional function

```
void proportional()
{
  int *x=IN_PORT, *y=OUT_PORT; // IO addres
  const k;
  *y=k(*x);
}
```

Integration over time

```
int a=0; // accumulator
void integrator()
{
  int *x=IN_PORT, *y=OUT_PORT; // IO addresse
  const dt; // run period
  a=a+(*x)*dt;
  *y=a;
}
```

- use float dt, if it is less than 1
- use integer frequency instead of dt, if $dt \ll 1$; then a=a+(*x)/freq

Derivatives

```
int old_x=0;
void deriv()
{
  int *x=IN_PORT, *y=OUT_PORT; // IO addresse
  const dt; // run period
  *y=( (*x) - old_x ) / dt;
  old_x=(*x);
}
```

- Danger! The derivative is sensitive to the discretization noise.
- One may want to save several old values and add a little more to the algorithm

Toggle

```
int tog=0;
void toggle()
{
  tog ^=1;
}
```

Sawtooth functions

```
int counter=0;
void sawtooth_rising()
{
  if (counter == modulus)
    counter=0;
  counter++;
}
```

- For the falling sawtooth use “counter -= 1”
- Raising-falling sawtooth – use toggle controlling an “if” statement to choose between rising and falling

PWM

```
int counter=0;
void pwm()
{
  int *y=OUT_PORT; // IO addresses
  counter++;
  if (counter == modulus)
    counter=0;
  if (counter < LEVEL)
    *y=1;
  else
    *y=0;
}
```

Arbitrary functions of time

```
int counter=0;
int wave_data[LENGTH]=
{...}; // define the function as a table
void waveform()
{
  int *y=OUT_PORT; // IO addresses
  *y = wave_data[counter];
  counter++;
}
```

PID controllers

- The mouse or the racing car project – one can use a single PIC to control several independent subsystems:
  - speed control
  - direction
  - “broomstick” balancing, if the mouse has two wheels
- Add to these several non-PID tasks
  - sensor data processing
  - labyrinth solving
- What is a PID controller? (you are supposed to know...)

Conclusions (revision)

- Real time vs. high-performance computing
- Taxonomy
- Principles of (illustrate with diagrams)
  - EDF preemptive ETS
  - EDF cooperative ETS
  - Timeline TTS
  - RM TTS
  - EDF TTS
- Optimality of EDF preemptive ETS/TTS, RM TTS – without proof
- Generic task model
- Timeline TTS – task/scheduler models, overruns, task algorithms