

CS162
Operating Systems and
Systems Programming
Lecture 9

Monitors (Continued)
Scheduling
Core Concepts and Classic Policies

Professor Natacha Crooks
<https://cs162.org/>

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatoiwicz,
Alison Norman and Lorenzo Alvisi

Recall: Monitors are better!

Use *locks* for mutual exclusion and *condition variables* for scheduling constraints

Monitor: a **lock** and zero or more **condition variables** for managing concurrent access to shared data

A monitor is a paradigm for concurrent programming

- Some languages like Java provide this natively
- Most others use actual locks and condition variables

Recall: Wait & Signal Pattern

```
...  
acquire(&buf_lock)  
...  
cond_signal(&buf_CV);  
...  
release(&buf_lock);
```

```
acquire(&buf_lock);  
...  
while (isEmpty(&queue)) {  
    cond_wait(&buf_CV, &buf_lock);  
}  
...  
lock.Release();
```

Hoare Semantics

Thread A

```
...  
acquire(&buf_lock)  
...  
cond_signal(&buf_CV);  
...  
release(&buf_lock);
```

Thread B

```
acquire(&buf_lock);  
...  
if (isEmpty(&queue)) {  
    cond_wait(&buf_CV, &buf_lock);  
}  
...  
lock.Release();
```

1. When call signal, handover buf_lock to thread B.
2. Thread B gets immediately scheduled (nothing can run in between).
3. Thread B eventually releases lock.

Mesa Semantics

Thread A

```
...  
acquire(&buf_lock)  
...  
cond_signal(&buf_CV);  
...  
release(&buf_lock);
```

Thread B

```
acquire(&buf_lock);  
...  
while (isEmpty(&queue)) {  
    cond_wait(&buf_CV, &buf_lock);  
}  
...  
lock.Release();
```

1. When call signal, keep lock. Place Thread B on **READY** queue (no special priority)
2. Thread A eventually releases buf_lock.
3. Thread B eventually gets scheduled and acquires buf_lock. Thread C may have run in between.
4. Thread B eventually releases buf_lock.

Basic Structure of *Mesa* Monitor Program

Monitors represent the synchronization logic of the program

- Wait if necessary
- Signal when change something so any waiting threads can proceed

```
lock
while (need to wait) {
    condvar.wait();
}
unlock
```

} Check and/or update state variables
Wait if necessary

do something so no need to wait

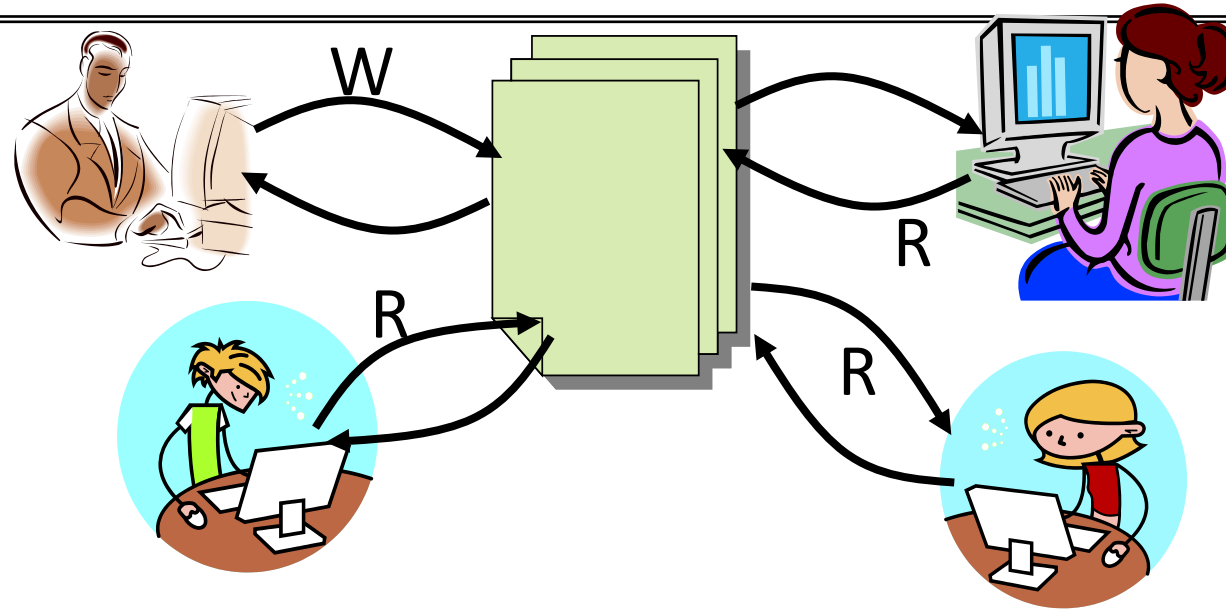
```
lock

condvar.signal();

unlock
```

} Check and/or update state variables

Readers/Writers Problem



Motivation: Consider a shared database

- Two classes of users:
 - » Readers - never modify database
 - » **W**riters - read and modify database
- Is using a single lock on the whole database sufficient?
 - » **L**ike to have many readers at the same time
 - » **O**nly one writer at a time

Basic Readers/Writers Solution

Correctness Constraints:

- Readers can access database when no writers
- Writers can access database when no readers or writers
- Only one thread manipulates state variables at a time

Basic structure of a solution:

- Reader ()

Wait until no writers

Access data base

Check out - wake up a waiting writer

- Writer ()

Wait until no active readers or writers

Access database

Check out - wake up waiting readers or writer

Basic Readers/Writers Solution

State variables (Protected by a lock called "lock"):

- » int **AR**: Number of active readers; initially = 0
- » int **WR**: Number of waiting readers; initially = 0
- » int **AW**: Number of active writers; initially = 0
- » int **WW**: Number of waiting writers; initially = 0
 - » Condition **okToRead** = NIL
 - » Condition **okToWrite** = NIL

Code for a Reader

```
Reader() {
    // First check self into system
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++;                // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--;                // No longer waiting
    }
    AR++;                    // Now we are active!
    release(&lock);

    // Perform actual read-only access
    AccessDatabase(ReadOnly);

    // Now, check out of system
    acquire(&lock);
    AR--;                    // No longer active
    if (AR == 0 && WW > 0) // No other active readers
        cond_signal(&okToWrite); // Wake up one writer
    release(&lock);
}
```

Code for a Writer

```
Writer() {
    // First check self into system
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; // Now we are active!
    release(&lock);
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    acquire(&lock);
    AW--; // No longer active
    if (WW > 0) { // Give priority to writers
        cond_signal(&okToWrite); // Wake up one writer
    } else if (WR > 0) { // Otherwise, wake reader
        cond_broadcast(&okToRead); // Wake all readers
    }
    release(&lock);
}
```

Simulation of Readers/Writers Solution

Use an example to simulate the solution

Consider the following sequence of operators:

-R1, R2, W1, R3

Initially: $AR = 0$, $WR = 0$, $AW = 0$,
 $WW = 0$

Simulation of Readers/Writers Solution

R1 comes along (no waiting threads)

AR = 0, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock)
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R1 comes along (no waiting threads)
AR = 0, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R1 comes along (no waiting threads)
AR = **1**, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R1 comes along (no waiting threads)

AR = **1**, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```


Simulation of Readers/Writers Solution

R1 accessing dbase (no other threads)

AR = 1, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}
```

AccessDBase(ReadOnly);

```
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R2 comes along (R1 accessing dbase)
 $AR = 1, WR = 0, AW = 0, WW = 0$

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R2 comes along (R1 accessing dbase)

AR = 1, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}
```

AccessDBase(ReadOnly);

```
acquire(&lock);
AR--;
if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
release(&lock);
}
```

Simulation of Readers/Writers Solution

R2 comes along (R1 accessing dbase)

AR = 2, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R2 comes along (R1 accessing dbase)

AR = 2, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
```

AccessDBase(ReadOnly);

```
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R1 and R2 accessing dbase

AR = 2, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}
```

AccessDBase(ReadOnly);

```
acquire(&lock);
AR--;
if (AR == 0 && WW > 0)
```

Assume readers take a while to access database
Situation: Locks released, only AR is non-zero

Simulation of Readers/Writers Solution

W1 comes along (R1 and R2 are still accessing dbase)

AR = 2, WR = 0, AW = 0, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;
    release(&lock);
}
```

AccessDBase(ReadWrite);

```
acquire(&lock);
AW--;
if (WW > 0) {
    cond_signal(&okToWrite);
} else if (WR > 0) {
    cond_broadcast(&okToRead);
}
release(&lock);
}
```

Simulation of Readers/Writers Solution

W1 comes along (R1 and R2 are still accessing dbase)

AR = 2, WR = 0, AW = 0, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;
    release(&lock);
}
```

AccessDBase(ReadWrite);

```
acquire(&lock);
AW--;
if (WW > 0) {
    cond_signal(&okToWrite);
} else if (WR > 0) {
    cond_broadcast(&okToRead);
}
release(&lock);
}
```


Simulation of Readers/Writers Solution

- W1 comes along (R1 and R2 are still accessing dbase)
- AR = 2, WR = 0, AW = 0, WW = 1

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No, Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;
    release(&lock);
}
```

AccessDBase(ReadWrite);

```
acquire(&lock);
AW--;
if (WW > 0) {
    cond_signal(&okToWrite);
} else if (WR > 0) {
    cond_broadcast(&okToRead);
}
release(&lock);
}
```

Simulation of Readers/Writers Solution

R3 comes along (R1 and R2 accessing dbase, W1 waiting)

$AR = 2$, $WR = 0$, $AW = 0$, $WW = 1$

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R3 comes along (R1 and R2 accessing dbase, W1 waiting)

$AR = 2, WR = 0, AW = 0, WW = 1$

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R3 comes along (R1 and R2 accessing dbase, W1 waiting)

AR = 2, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
}
```

AccessDBase(ReadOnly);

```
acquire(&lock);
AR--;
if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
release(&lock);
}
```

Simulation of Readers/Writers Solution

R3 comes along (R1, R2 accessing dbase, W1 waiting)

AR = 2, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}
```

AccessDBase(ReadOnly);

```
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R1 and R2 accessing dbase, W1 and R3 waiting

AR = 2, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}
```

AccessDBase(ReadOnly);

```
acquire(&lock);
AR--;
if (AR == 0 && WW > 0)
```

Status:

- R1 and R2 still reading
- W1 and R3 waiting on okToWrite and okToRead, respectively

Simulation of Readers/Writers Solution

R2 finishes (R1 accessing dbase, W1 and R3 waiting)

AR = 2, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}
```

AccessDBase(ReadOnly);

```
acquire(&lock);
AR--;
if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
release(&lock);
}
```

Simulation of Readers/Writers Solution

R2 finishes (R1 accessing dbase, W1 and R3 waiting)

AR = 1, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```


Simulation of Readers/Writers Solution

R2 finishes (R1 accessing dbase, W1 and R3 waiting)

AR = 1, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R2 finishes (R1 accessing dbase, W1 and R3 waiting)

AR = 1, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}
```

AccessDBase(ReadOnly);

```
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R1 finishes (W1 and R3 waiting)

AR = 1, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}
```

AccessDBase(ReadOnly);

```
acquire(&lock);
AR--;
if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
release(&lock);
}
```

Simulation of Readers/Writers Solution

R1 finishes (W1, R3 waiting)

AR = 0, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R1 finishes (W1, R3 waiting)

AR = 0, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R1 signals a writer (W1 and R3 waiting)

AR = 0, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

W1 gets signal (R3 still waiting)
AR = 0, WR = 1, AW = 0, WW = 1

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No, Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;
    release(&lock);
}
```

AccessDBase(ReadWrite);

```
acquire(&lock);
AW--;
if (WW > 0) {
    cond_signal(&okToWrite);
} else if (WR > 0) {
    cond_broadcast(&okToRead);
}
release(&lock);
}
```

Simulation of Readers/Writers Solution

W1 gets signal (R3 still waiting)
AR = 0, WR = 1, AW = 0, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;
    release(&lock);
}
```

AccessDBase(ReadWrite);

```
acquire(&lock);
AW--;
if (WW > 0) {
    cond_signal(&okToWrite);
} else if (WR > 0) {
    cond_broadcast(&okToRead);
}
release(&lock);
}
```


Simulation of Readers/Writers Solution

W1 gets signal (R3 still waiting)
AR = 0, WR = 1, AW = 1, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;
    release(&lock);
}
```

AccessDBase(ReadWrite);

```
acquire(&lock);
AW--;
if (WW > 0) {
    cond_signal(&okToWrite);
} else if (WR > 0) {
    cond_broadcast(&okToRead);
}
release(&lock);
}
```

Simulation of Readers/Writers Solution

W1 accessing dbase (R3 still waiting)
AR = 0, WR = 1, AW = 1, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;
    release(&lock);
}
```

AccessDBase(ReadWrite);

```
acquire(&lock);
AW--;
if (WW > 0) {
    cond_signal(&okToWrite);
} else if (WR > 0) {
    cond_broadcast(&okToRead);
}
release(&lock);
}
```

Simulation of Readers/Writers Solution

W1 finishes (R3 still waiting)

AR = 0, WR = 1, AW = 1, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;
    release(&lock);
}
```

AccessDBase(ReadWrite);

```
acquire(&lock);
AW--;
if (WW > 0) {
    cond_signal(&okToWrite);
} else if (WR > 0) {
    cond_broadcast(&okToRead);
}
release(&lock);
}
```

Simulation of Readers/Writers Solution

W1 finishes (R3 still waiting)

AR = 0, WR = 1, AW = 0, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;
    release(&lock);
}
```

AccessDBase(ReadWrite);

```
    acquire(&lock);
    AW--;
    if (WW > 0) {
        cond_signal(&okToWrite);
    } else if (WR > 0) {
        cond_broadcast(&okToRead);
    }
    release(&lock);
}
```

Simulation of Readers/Writers Solution

W1 finishes (R3 still waiting)

AR = 0, WR = 1, AW = 0, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;
    release(&lock);
}
```

AccessDBase(ReadWrite);

```
    acquire(&lock);
    AW--;
    if (WW > 0) {
        cond_signal(&okToWrite);
    } else if (WR > 0) {
        cond_broadcast(&okToRead);
    }
    release(&lock);
}
```

Simulation of Readers/Writers Solution

W1 signaling readers (R3 still waiting)
AR = 0, WR = 1, AW = 0, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++;
    release(&lock);
}
```

AccessDBase(ReadWrite);

```
acquire(&lock);
AW--;
if (WW > 0) {
    cond_signal(&okToWrite);
} else if (WR > 0) {
    cond_broadcast(&okToRead);
}
release(&lock);
}
```

Simulation of Readers/Writers Solution

R3 gets signal (no waiting threads)
AR = 0, WR = 1, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R3 gets signal (no waiting threads)

AR = 0, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```


Simulation of Readers/Writers Solution

R3 accessing dbase (no waiting threads)
AR = 1, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}
```

AccessDBase(ReadOnly);

```
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Simulation of Readers/Writers Solution

R3 finishes (no waiting threads)

$AR = 1, WR = 0, AW = 0, WW = 0$

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
}
```

AccessDBase(ReadOnly);

```
acquire(&lock);
AR--;
if (AR == 0 && WW > 0)
    cond_signal(&okToWrite);
release(&lock);
}
```

Simulation of Readers/Writers Solution

R3 finishes (no waiting threads)

AR = 0, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);

    AccessDbase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        cond_signal(&okToWrite);
    release(&lock);
}
```

Questions

Can readers starve? Consider Reader() entry code:

```
while ((AW + WW) > 0) { // Is it safe to read?
    WR++;                // No. Writers exist
    cond_wait(&okToRead, &lock); // Sleep on cond var
    WR--;                // No longer waiting
}
AR++;                    // Now we are active!
```

What if we erase the condition check in Reader exit?

```
AR--;                    // No longer active
if (AR == 0 && WW > 0) // No other active readers
    cond_signal(&okToWrite); // Wake up one writer
```

Questions

Further, what if we turn the `signal()` into `broadcast()`

```
AR--; // No longer active
cond_broadcast(&okToWrite); // Wake up sleepers
```

Finally, what if we use only one condition variable (call it “`okContinue`”) instead of two separate ones?

- Both readers and writers sleep on this variable
- Must use `broadcast()` instead of `signal()`

Code for a Reader

```
Reader() {
    // First check self into system
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++;                // No. Writers exist
        cond_wait(&okToRead, &lock); // Sleep on cond var
        WR--;                // No longer waiting
    }
    AR++;                    // Now we are active!
    release(&lock);

    // Perform actual read-only access
    AccessDatabase(ReadOnly);

    // Now, check out of system
    acquire(&lock);
    AR--;                    // No longer active
    if (AR == 0 && WW > 0) // No other active readers
        cond_signal(&okToWrite); // Wake up one writer
    release(&lock);
}
```

Code for a Writer

```
Writer() {
    // First check self into system
    acquire(&lock);
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        cond_wait(&okToWrite, &lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; // Now we are active!
    release(&lock);
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    acquire(&lock);
    AW--; // No longer active
    if (WW > 0) { // Give priority to writers
        cond_signal(&okToWrite); // Wake up one writer
    } else if (WR > 0) { // Otherwise, wake reader
        cond_broadcast(&okToRead); // Wake all readers
    }
    release(&lock);
}
```

Mesa Monitor Conclusion

Monitors represent the synchronization logic of the program

- Wait if necessary
- Signal when change something so any waiting threads can proceed

lock

```
while (need to wait) {  
    condvar.wait();  
}
```



Check and/or update
state variables
Wait if necessary

unlock

do something so no need to wait

lock

```
condvar.signal();
```



Check and/or update
state variables

unlock

C-Language Support for Synchronization

C language: Pretty straightforward synchronization

Just make sure you know *all* the code paths out
of a critical section

```
int Rtn() {
    acquire(&lock);
    ...
    if (exception) {
        release(&lock);
        return errReturnCode;
    }
    ...
    release(&lock);
    return OK;
}
```

Concurrency and Synchronization in C

Harder with more locks

```
void Rtn() {
    lock1.acquire();
    ...
    if (error) {
        lock1.release();
        return;
    }
    ...
    lock2.acquire();
    ...
    if (error) {
        lock2.release();
        lock1.release();
        return;
    }
    ...
    lock2.release();
    lock1.release();
}
```

C++ Language Support for Synchronization

Languages with exceptions like C++
-Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)

```
void Rtn() {  
    lock.acquire();  
    ...  
    DoFoo();  
    ...  
    lock.release();  
}  
void DoFoo() {  
    ...  
    if (exception) throw errException;  
    ...  
}
```

-Notice that an exception in DoFoo() will exit without releasing the lock!

C++ Language Support for Synchronization (con't)

Must catch all exceptions in critical sections

- Catch exceptions, release lock, and re-throw exception:

```
void Rtn() {
    lock.acquire();
    try {
        ...
        DoFoo();
        ...
    } catch (...) {           // catch exception
        lock.release();      // release lock
        throw;                // re-throw the exception
    }
    lock.release();
}
void DoFoo() {
    ...
    if (exception) throw errException;
    ...
}
```

Much better: C++ Lock Guards

```
#include <mutex>
int global_i = 0;
std::mutex global_mutex;

void safe_increment() {
    std::lock_guard<std::mutex> lock(global_mutex);
    ...
    global_i++;
    // Mutex released when 'lock' goes out of scope
}
```

Python with Keyword

More versatile than we show here (can be used to close files, database connections, etc.)

```
lock = threading.Lock()
```

```
...
```

```
with lock: # Automatically calls acquire()
```

```
    some_var += 1
```

```
...
```

```
# release() called however we leave block
```

Java synchronized Keyword

Every Java object has an associated lock:

- Lock is acquired on entry and released on exit from a `synchronized` method
- Lock is properly released if exception occurs inside a `synchronized` method
- Mutex execution of `synchronized` methods (beware deadlock)

```
class Account {
    private int balance;

    // object constructor
    public Account (int initialBalance) {
        balance = initialBalance;
    }
    public synchronized int getBalance() {
        return balance;
    }
    public synchronized void deposit(int amount) {
        balance += amount;
    }
}
```

Java Support for Monitors

Along with a lock, every object has a single condition variable associated with it

To wait inside a synchronized method:

- void wait();
- void wait(long timeout);

To signal while in a synchronized method:

- void notify();
- void notifyAll();

Where are we going with synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Compare&Swap

Implement various higher-level synchronization primitives using atomic operations

Topic Breakdown

Virtualizing the CPU

Process Abstraction and API

Threads and Concurrency

Scheduling

Virtualizing Memory

Virtual Memory

Paging

Persistence

IO devices

File Systems

Distributed Systems

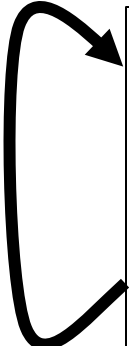
Challenges with distribution

Data Processing & Storage

Goals for Today

- **What is scheduling?**
- **What makes a good scheduling policy?**
- **What are existing schedulers and how do they perform?**

The Scheduling Loop!



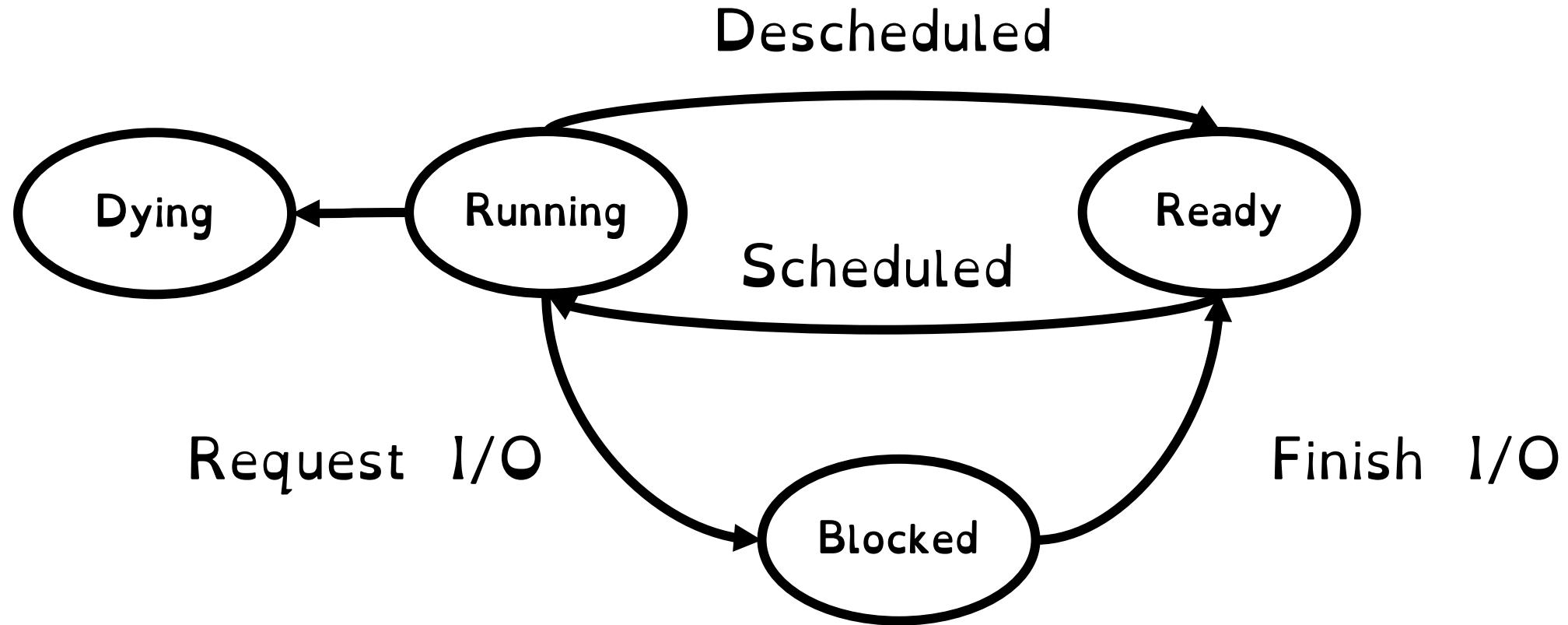
```
if (readyThreads(TCBs) ) {
    nextTCB = selectThread(TCBs);
    run(nextTCB);
} else {
    run_idle_thread();
}
```

1. Which task to run next?

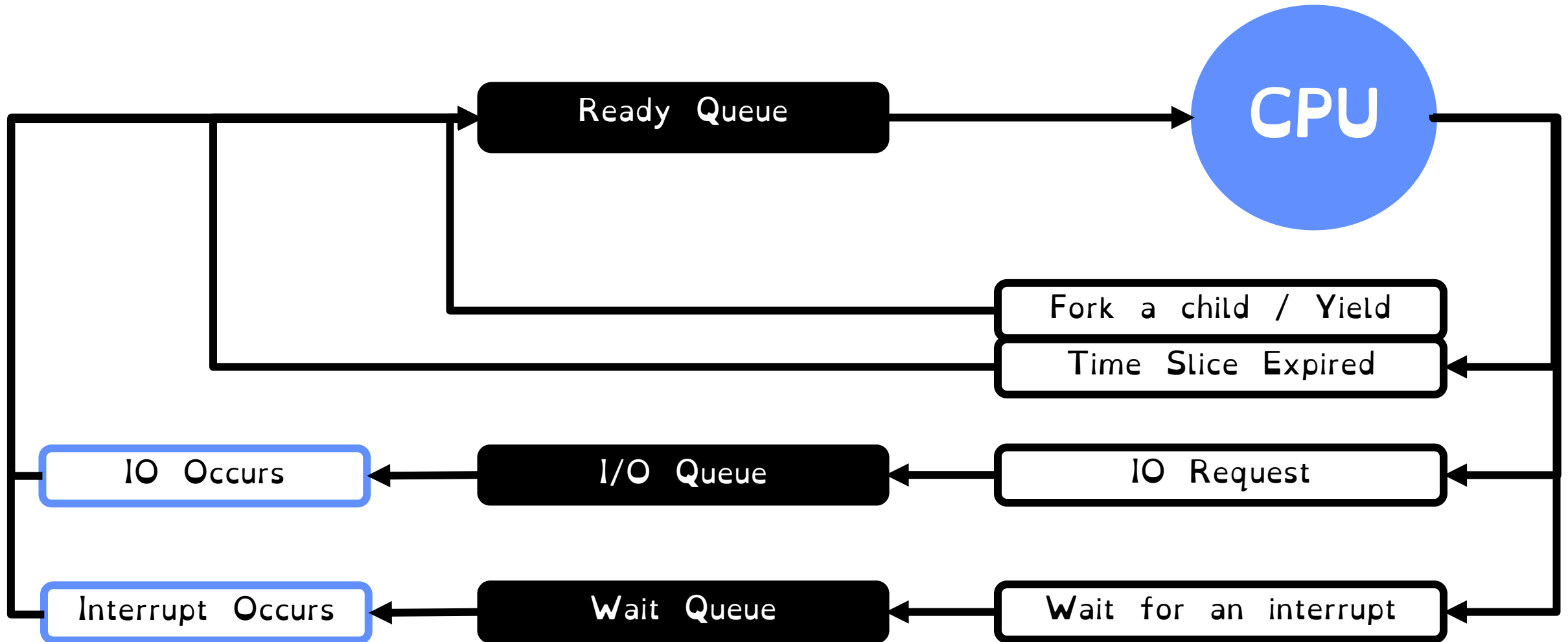
2. How frequently does this loop run?

3. What happens if run never returns?

Recall: Thread Life Cycle



Recall: What triggers a scheduling decision?



What makes a good scheduling policy?

A hopeless Queue.

The Queue For the UK Queen

6 miles (10 KM) long.

Visible from Space.

A bad but more realistic queue.

The DMV

What makes a good scheduling policy?

What does the DMV care about?



What do individual users care about?



Important Performance Metrics

Response time (or latency).

User-perceived time to do some task

Throughput.

The rate at which tasks are completed

Scheduling overhead.

The time to switch from one task to another.

Predictability.

Variance in response times for repeated requests.

Important Performance Metrics

Fairness

Equality in the performance perceived by one task

Starvation

The lack of progress for one task, due to resources being allocated to different tasks

Sample Scheduling Policies

Assume DMV job A takes 1 second, job B takes 2 days

Policy Idea: Only ever schedule users with Job A

What is the metric we are optimizing?

A) Throughput B) Latency C) Predictability D) Low-Overhead

Can the schedule lead to starvation?

A) Yes B) No

Is the schedule fair?

A) Yes B) No

Sample Scheduling Policies

Assume DMV consists only of jobs of type A.

Policy Idea: Schedule jobs randomly

What is the metric we are optimizing?

A) Throughput B) Latency C) Predictability D) Low-Overhead

Can the schedule lead to starvation?

A) Yes B) No

Is the schedule fair?

A) Yes B) No

Sample Scheduling Policies

Assume DMV consists only of 100 different types of jobs. Some jobs need Clerk A, some Clerks A&B, others Clerk C.

Policy Idea Every time schedule a job, compute all possible orderings of jobs, pick one that finishes quickest

What is the metric we are optimizing?

A) Throughput B) Latency C) Predictability D) Low-Overhead

Can the schedule lead to starvation?

A) Yes B) No

Is the schedule fair?

A) Yes B) No

Scheduling Policy Goals/Criteria

Minimise
Response Time

Maximise
Throughput

While remaining fair and starvation-free

Useful metrics

Waiting time for P

Total Time spent waiting for *CPU*

Average waiting time

Average of all processes' wait time

Response Time for P

Time to when process gets first scheduled

Completion time

Waiting time + Run time

Average completion time

Average of all processes' completion time

Assumptions

Threads are independent!

One thread = One User

Unrealistic but simplify the problem
so it can be solved

Only look at **work-conserving** scheduler
=> Never leave processor idle if work to do

Workload Assumptions

A workload is a set of tasks for some system to perform, including how long tasks last and when they arrive

Compute-Bound

Tasks that primarily perform compute

Fully utilise CPU

IO Bound

Mostly wait for IO, limited compute

Often in the Blocked state

First-Come, First-Served (FCFS)

Run tasks in order of arrival.

Run task until completion (or blocks on IO).
No preemption

This is the DMV model.

Also called FIFO

First-Come, First-Served (FCFS)

Process

Burst Time

P_1

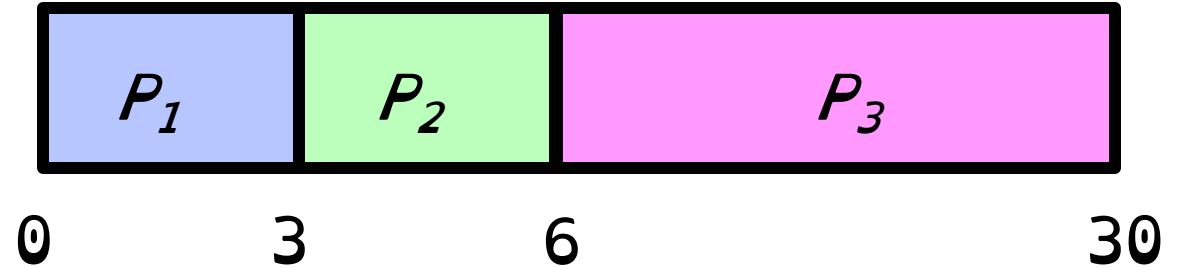
3

P_2

3

P_3

24



What is the average completion time? $(\frac{3+6+30}{3} = 13)$

What is the average waiting time? $(\frac{0+3+6}{3} = 3)$

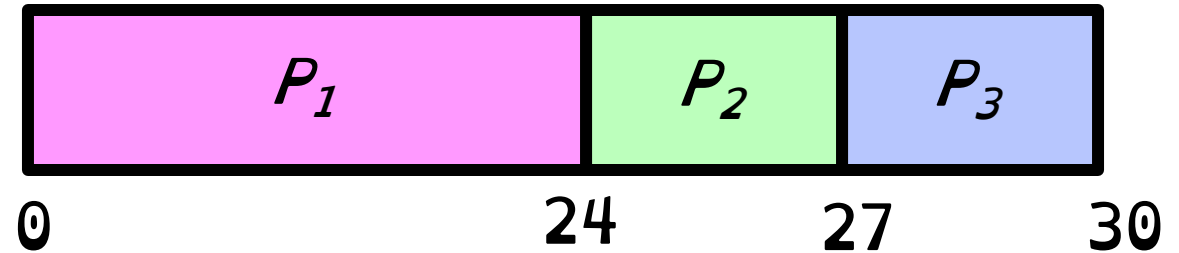
First-Come, First-Served (FCFS)

Process Burst Time

P_3 24

P_2 3

P_1 3



What is the average completion time? $(\frac{24+27+30}{3} = 27)$

What is the average waiting time? $(\frac{0+24+27}{3} = 17)$

The Convoy Effect

FIFO/FCFS very sensitive to arrival order

Convoy effect

Short process stuck behind long process
Lots of small tasks build up behind long tasks
FIFO is *non-preemptible*

The Convoy Effect

FIFO/FCFS very sensitive to arrival order

Convoy effect

Short process stuck behind long process
Lots of small tasks build up behind long tasks
FIFO is *non-preemptible*



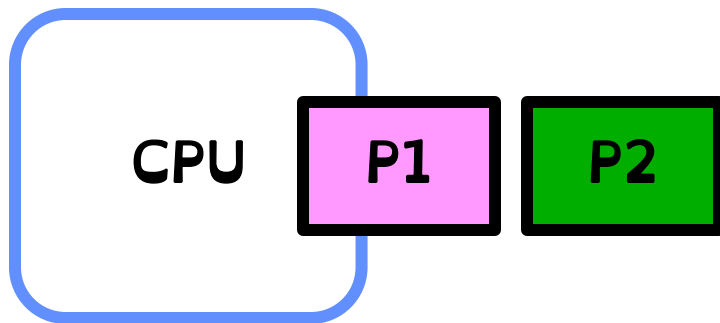
The Convoy Effect

FIFO/FCFS very sensitive to arrival order

Convoy effect

Short process stuck behind long process
Lots of small tasks build up behind long tasks

FIFO is *non-preemptible*

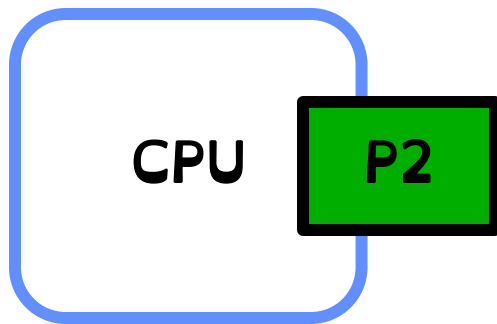


The Convoy Effect

FIFO/FCFS very sensitive to arrival order

Convoy effect

Short process stuck behind long process
Lots of small tasks build up behind long tasks
FIFO is *non-preemptible*

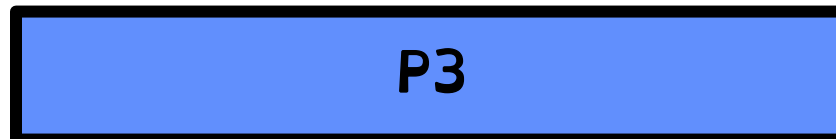
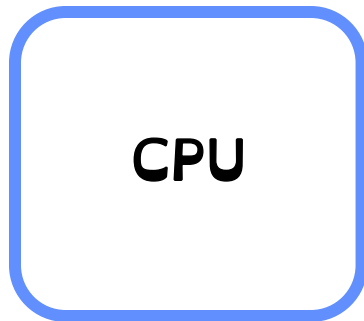


The Convoy Effect

FIFO/FCFS very sensitive to arrival order

Convoy effect

Short process stuck behind long process
Lots of small tasks build up behind long tasks
FIFO is *non-preemptible*

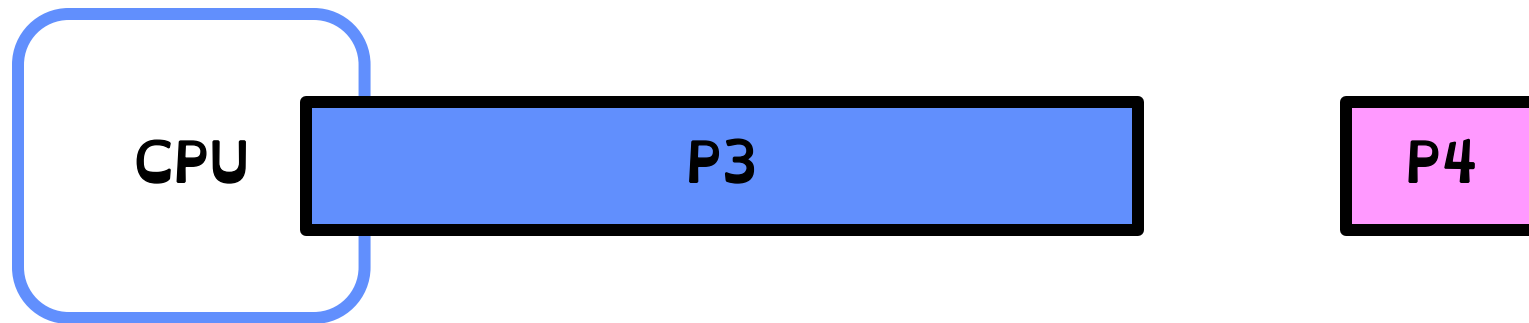


The Convoy Effect

FIFO/FCFS very sensitive to arrival order

Convoy effect

Short process stuck behind long process
Lots of small tasks build up behind long tasks
FIFO is *non-preemptible*

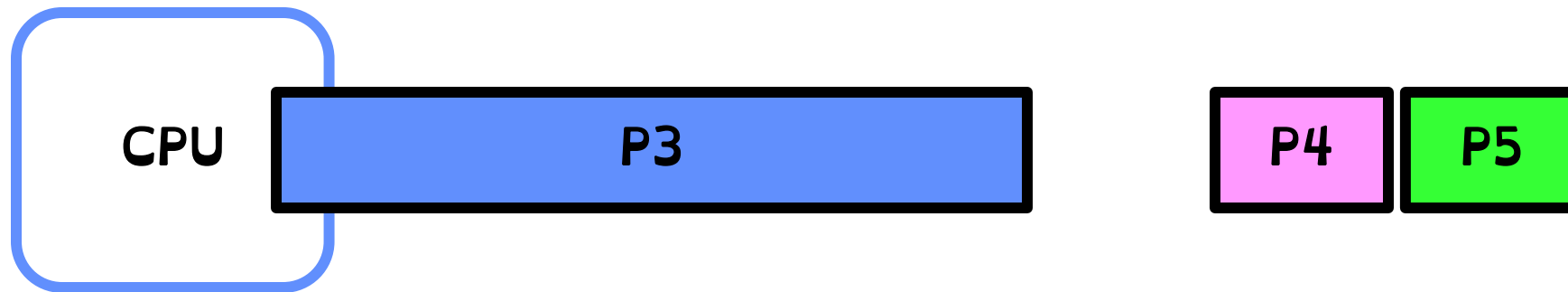


The Convoy Effect

FIFO/FCFS very sensitive to arrival order

Convoy effect

Short process stuck behind long process
Lots of small tasks build up behind long tasks
FIFO is *non-preemptible*

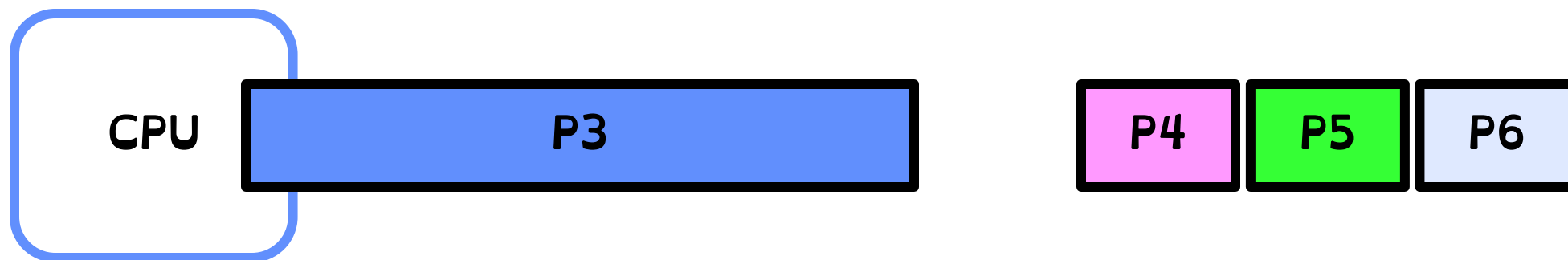


The Convoy Effect

FIFO/FCFS very sensitive to arrival order

Convoy effect

Short process stuck behind long process
Lots of small tasks build up behind long tasks
FIFO is *non-preemptible*



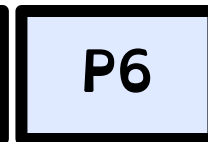
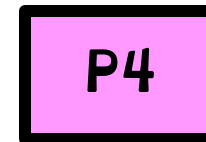
The Convoy Effect

FIFO/FCFS very sensitive to arrival order

Convoy effect

Short process stuck behind long process
Lots of small tasks build up behind long tasks
FIFO is *non-preemptible*

Can FIFO lead to starvation?



FCFS/FIFO Summary

The good

Simple
Low Overhead
No Starvation

The bad

Sensitive to arrival
order (poor
predictability)

The ugly

Convoy Effect.
Bad for Interactive
Tasks

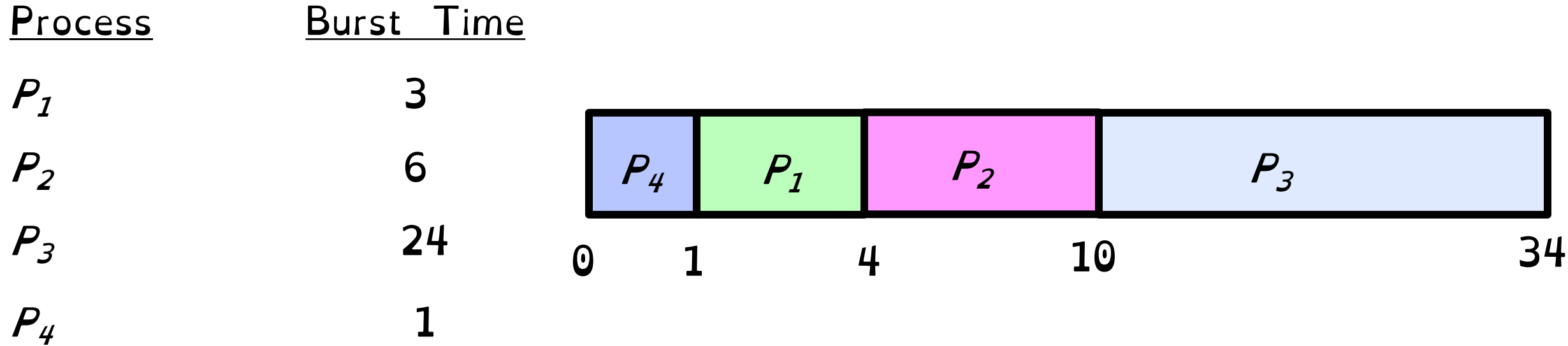
Shortest Job First

How can we minimise average completion time?

By scheduling jobs in order of
estimated completion time

This is the “10 items or less” line at Safeway

Shortest Job First



What is the average completion time? $(\frac{1+4+10+34}{4} = 12.25)$

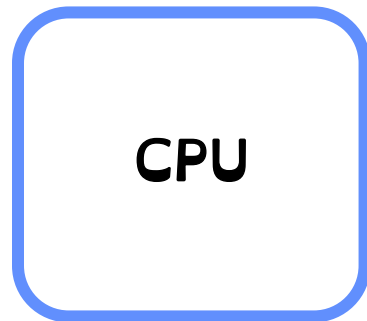
Can prove that SJF generates optimal average completion time if all jobs arrive at the same time

Are we done?

Can SJF lead to starvation?

Yes

Any scheduling policy that always favours a **fixed property** for scheduling leads to starvation

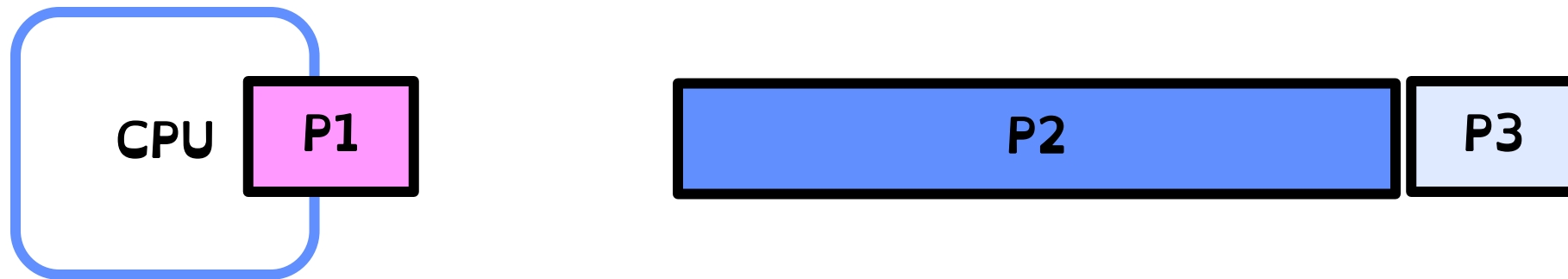


Are we done?

Can SJF lead to starvation?

Yes

Any scheduling policy that always favours a **fixed property** for scheduling leads to starvation

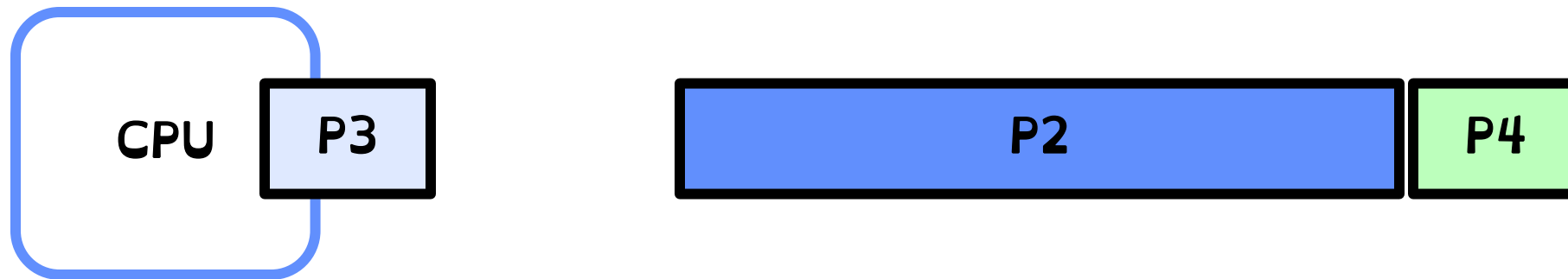


Are we done?

Can SJF lead to starvation?

Yes

Any scheduling policy that always favours a **fixed property** for scheduling leads to starvation

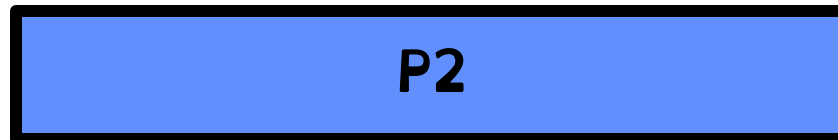
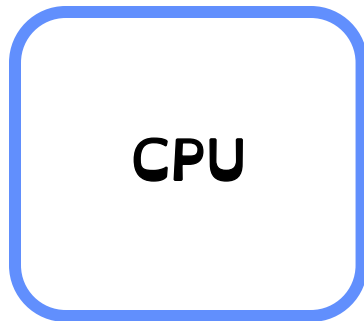


Are we done?

Is SFJ subject to the convoy effect?

Yes

Any **non-preemptible** scheduling policy suffers from convoy effect

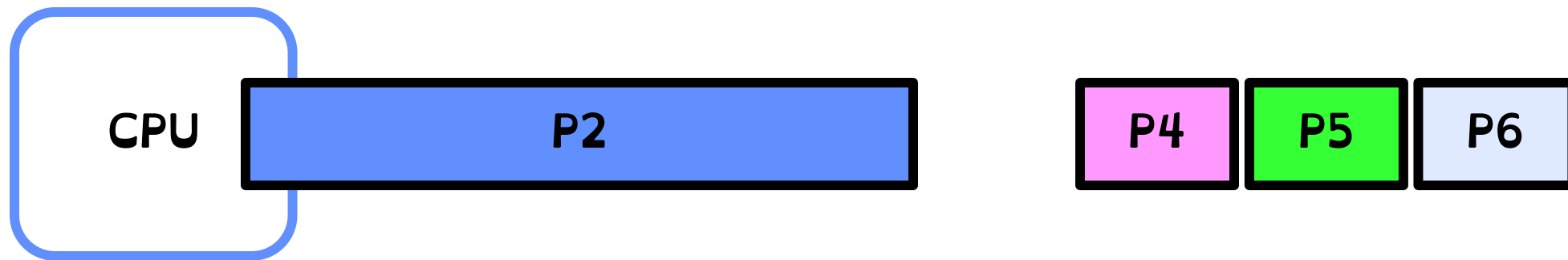


Are we done?

Is SFJ subject to the convoy effect?

Yes

Any **non-preemptible** scheduling policy suffers from convoy effect



SJF Summary

The good

Optimal **Average**
Completion Time when
jobs arrive
simultaneously

The bad

Sensitive to arrival
order (poor
predictability)

The ugly

Can lead to starvation!

Requires knowing duration of job

Shortest Time to Completion First (STCF)

Introduce the notion of **preemption**

A running task can be de-scheduled before completion.

STCF

Schedule the task with the **least amount of time left**

Shortest Time to Completion First (STCF)

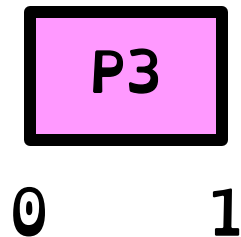
STCF

Schedule the task with the least amount of time left

<u>Process</u>	<u>Burst Time (left)</u>	<u>Arrival Time</u>
P_1	3	10
P_2	6	1
P_3	24	0
P_4	16	20

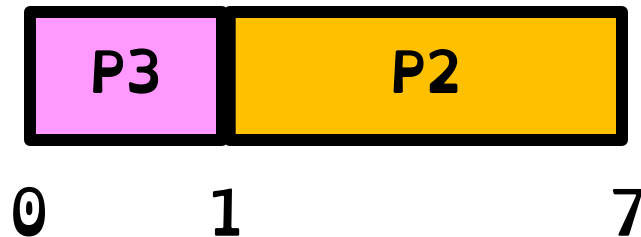
Shortest Time to Completion First (STCF)

<u>Process</u>	<u>Burst Time (left)</u>	<u>Arrival Time</u>
P_1	3	10
P_2	6	1
P_3	24	0
P_4	16	18



Shortest Time to Completion First (STCF)

<u>Process</u>	<u>Burst Time (left)</u>	<u>Arrival Time</u>
P_1	3	10
P_2	6	1
P_3	23	0
P_4	16	18



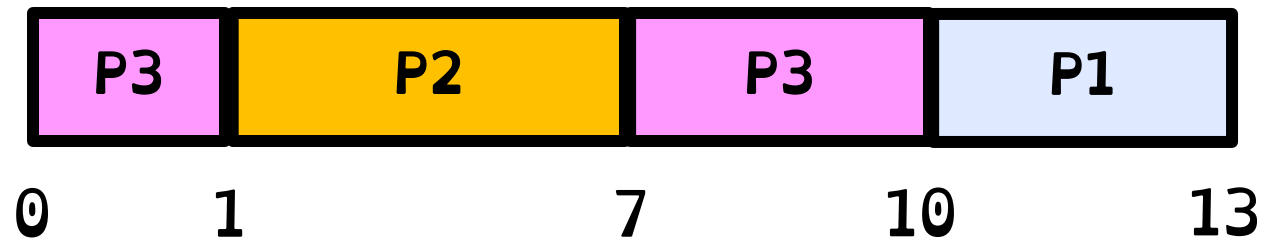
Shortest Time to Completion First (STCF)

<u>Process</u>	<u>Burst Time (left)</u>	<u>Arrival Time</u>
P_1	3	10
P_2	0	1
P_3	23	0
P_4	16	20



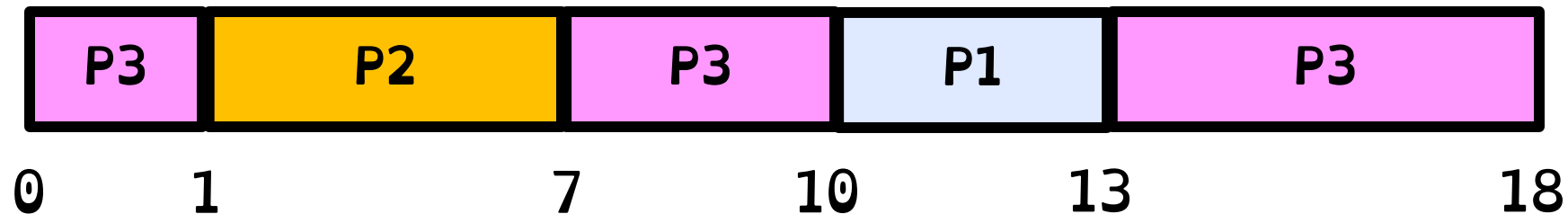
Shortest Time to Completion First (STCF)

<u>Process</u>	<u>Burst Time (left)</u>	<u>Arrival Time</u>
P_1	3	10
P_2	0	1
P_3	20	0
P_4	16	18



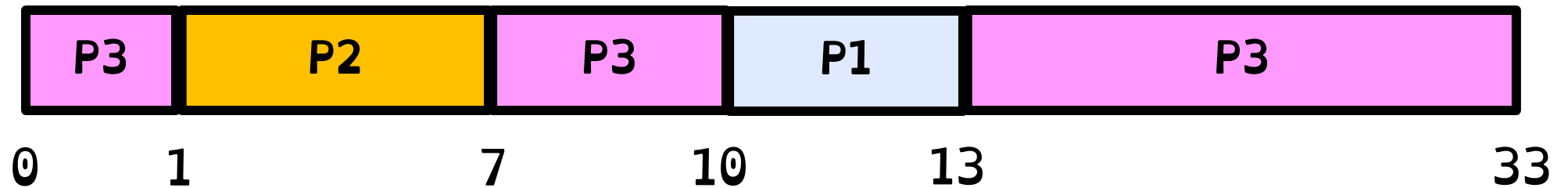
Shortest Time to Completion First (STCF)

<u>Process</u>	<u>Burst Time (left)</u>	<u>Arrival Time</u>
P_1	0	10
P_2	0	1
P_3	15	0
P_4	16	18



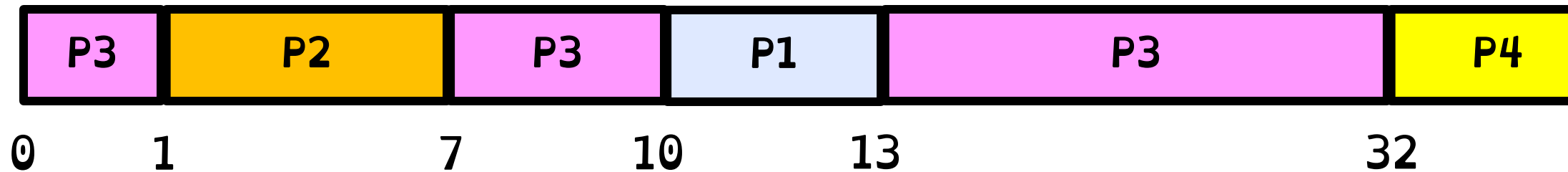
Shortest Time to Completion First (STCF)

<u>Process</u>	<u>Burst Time (left)</u>	<u>Arrival Time</u>
P_1	0	10
P_2	0	1
P_3	0	0
P_4	15	18



Shortest Time to Completion First (STCF)

<u>Process</u>	<u>Burst Time (left)</u>	<u>Arrival Time</u>
P_1	0	10
P_2	0	1
P_3	0	0
P_4	15	18



Are we done?

Can STCF lead to starvation?

Yes

Any scheduling policy that
always favours a **fixed property**
for scheduling leads starvation

No change!

Are we done?

Is STCF subject to the convoy effect?

No!

STCF is a preemptible policy

STCF Summary

The good

Optimal Average
Completion Time Always

The bad

The ugly

Can lead to starvation!

Requires knowing duration of job

Taking a step back

Property	FCFS	SJF	STCF
Optimise Average Completion Time		✓	✓
Prevent Starvation	✓		
Prevent Convoy Effect			✓
Psychic Skills Not Needed	✓		

Can we design a non-psychic, starvation-free scheduler with good response time?

Round-Robin Scheduling

RR runs a job for a time slice
(a scheduling quantum)

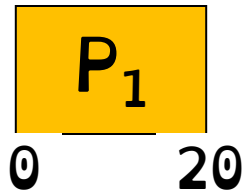
Once time slice over,
Switch to next job in ready queue.
=> Called time-slicing

RR with Time Quantum = 20

<u>Process</u>	<u>Burst Time</u>
P_1	53
P_2	8
P_3	68
P_4	24

RR with Time Quantum = 20

<u>Process</u>	<u>Burst Time</u>
P_1	53 => 33
P_2	8
P_3	68
P_4	24



RR with Time Quantum = 20

Process

P_1

P_2

P_3

P_4

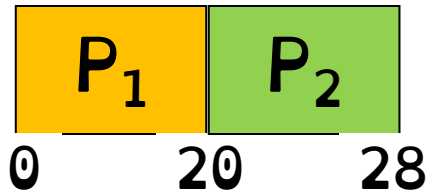
Burst Time

33

8 \Rightarrow 0

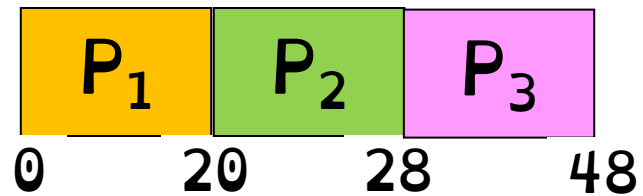
68

24



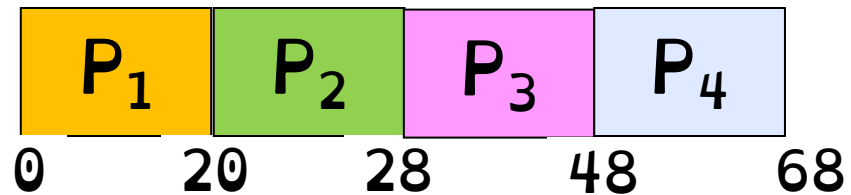
RR with Time Quantum = 20

<u>Process</u>	<u>Burst Time</u>
P_1	33
P_2	0
P_3	68 \Rightarrow 48
P_4	24



RR with Time Quantum = 20

<u>Process</u>	<u>Burst Time</u>
P_1	33
P_2	0
P_3	48
P_4	24 \Rightarrow 4



RR with Time Quantum = 20

Process

P_1

P_2

P_3

P_4

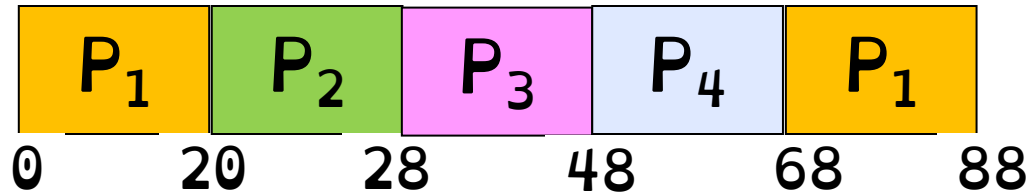
Burst Time

33 \Rightarrow 13

0

48

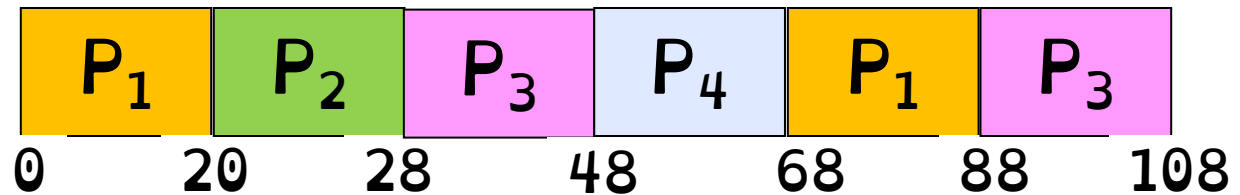
4



RR with Time Quantum = 20

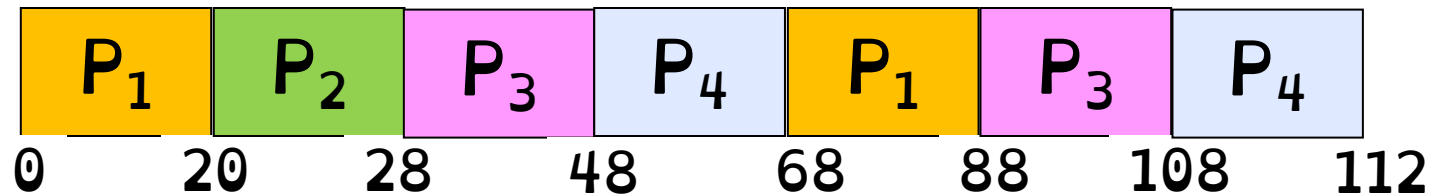
<u>Process</u>	<u>Burst Time</u>
P_1	13
P_2	0
P_3	48
P_4	4

=> 28



RR with Time Quantum = 20

<u>Process</u>	<u>Burst Time</u>
P_1	13
P_2	0
P_3	28
P_4	4 => 0



RR with Time Quantum = 20

Waiting time

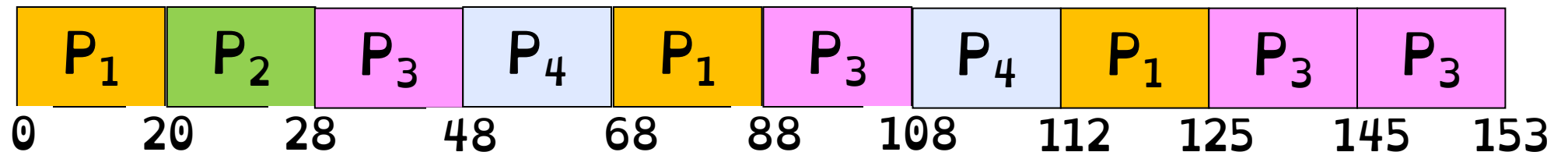
- $P_1 = 0 + (68-20) + (112-88) = 72$
- $P_2 = (20-0) = 20$
- $P_3 = (28-0) + (88-48) + (125-108) + 0 = 85$
- $P_4 = (48-0) + (108-68) = 88$

Average waiting time

$$\left(\frac{72+20+85+88}{4} = 66.25 \right)$$

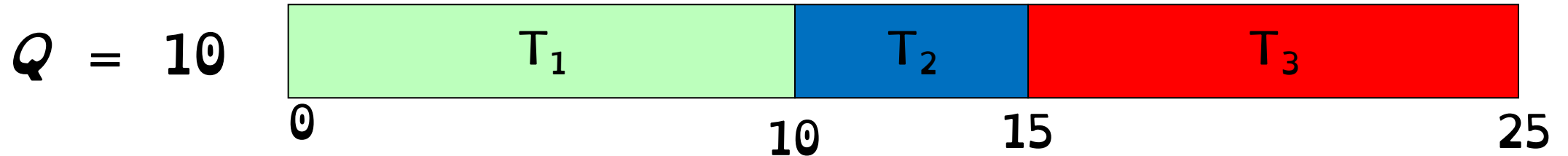
Average completion time

$$\left(\frac{125+28+153+112}{4} = 104.25 \right)$$

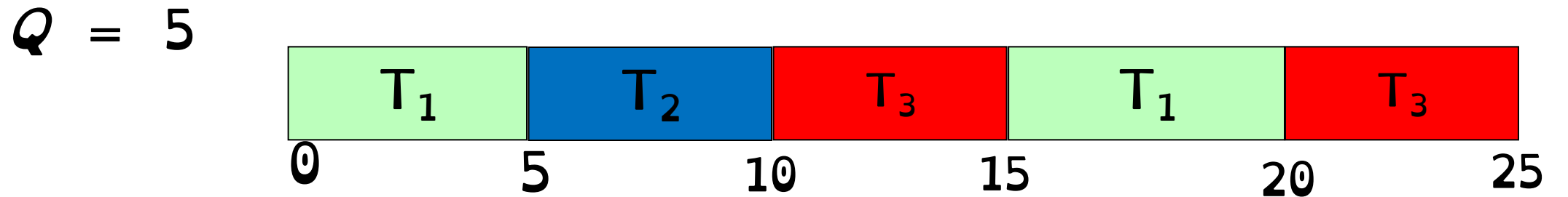


Decrease Completion Time

- T_1 : Burst Length 10
- T_2 : Burst Length 5
- T_3 : Burst Length 10



$$\text{Average Completion Time} = (10 + 15 + 25)/3 = 16.7$$



$$\text{Average Completion Time} = (20 + 10 + 25)/3 = 18.3$$

Switching is not free!

Small scheduling quantas lead to frequent context switches

- Mode switch overhead
- Trash cache-state

q must be large with respect to context switch, otherwise overhead is too high

Are we done?

Can RR lead to starvation?

No

No process waits more than $(n-1)q$ time units

Are we done?

Can RR suffer from convoy effect?

No

Only run a time-slice at a time

RR Summary

The good

Bounded response time

The bad

Completion time can be high (stretches out long jobs)

The ugly

Overhead of context switching

Taking a step back

Property	FCFS	SJF	STCF
Optimise Average Completion Time		✓	✓
Prevent Starvation	✓		
Prevent Convoy Effect			✓
Psychic Skills Not Needed	✓		

Taking a step back

Property	FCFS	SJF	STCF	RR
Optimise Average Completion Time		✓	✓	
Optimise Average Response Time				✓
Prevent Starvation	✓			✓
Prevent Convoy Effect			✓	✓
Psychic Skills Not Needed	✓			✓

FCFS and Round Robin Showdown

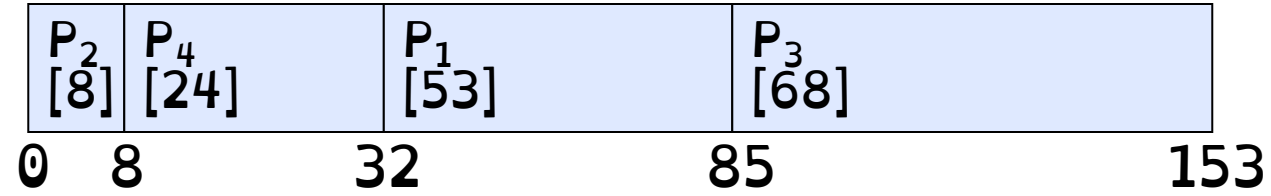
Assuming zero-cost context-switching time,
is RR always better than FCFS?

10 jobs, each take 100s of CPU time
RR scheduler quantum of 1s
All jobs start at the same time

Job #	FIFO	RR
1	100	991
2	200	992
...
9	900	999
10	1000	1000

Earlier Example with Different Time Quantum

Best FCFS:



Quantum	P1	P2	P3	P4	Average
Best FCFS	85	8	16	32	69.5
Q=1	137	30	153	81	100.5
Q=5	135	28	153	82	99.5
Q=8	133	16	153	80	99,5
Q=10	135	18	153	92	104.5
Q=20	125	28	153	112	104.5
Worst FCFS	121	153	68	145	121.75