Concurrent Computing

Lecturers: Prof. Majid Mirmehdi majid@cs.bris.ac.uk
Dr. Tilo Burghardt tilo@cs.bris.ac.uk
Dr. Daniel Page page@cs.bris.ac.uk

Web: http://www.cs.bris.ac.uk/Teaching/Resources/COMS20001

LECTURE MM3

REPLICATION AND PIPELINING

Process Replication using Replicated PAR

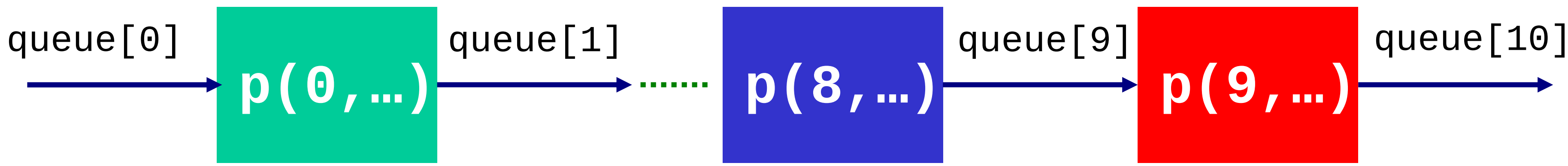
- builds an array of **structurally similar** parallel processes
- parameterise processes using the replicator index (e.g. **`i`**)
- **Example**: use together with channels and arrays for process chains, buffers, queues etc.

```
// chain of processes

void chainElement( chanend cInput, chanend cOutput) { ... }

int main ( void ) {
  chan c[4];
   par (int i=0; i<3; i++)
     chainElement(c[i], c[i+1]);
return 0;
}
```

Replicated **PAR** is key to elegant concurrent programming!

Example: Process Chain via Replicated **PAR**

```
...
  chan queue[11];                    // array of 11 channels
  par (int i=0; i<10; i++) {  // set up 10 parallel processes p in chain
    int a;
    queue[i] :> a;
    p(i,a);
    queue[i+1] <: a;
  }
...
```



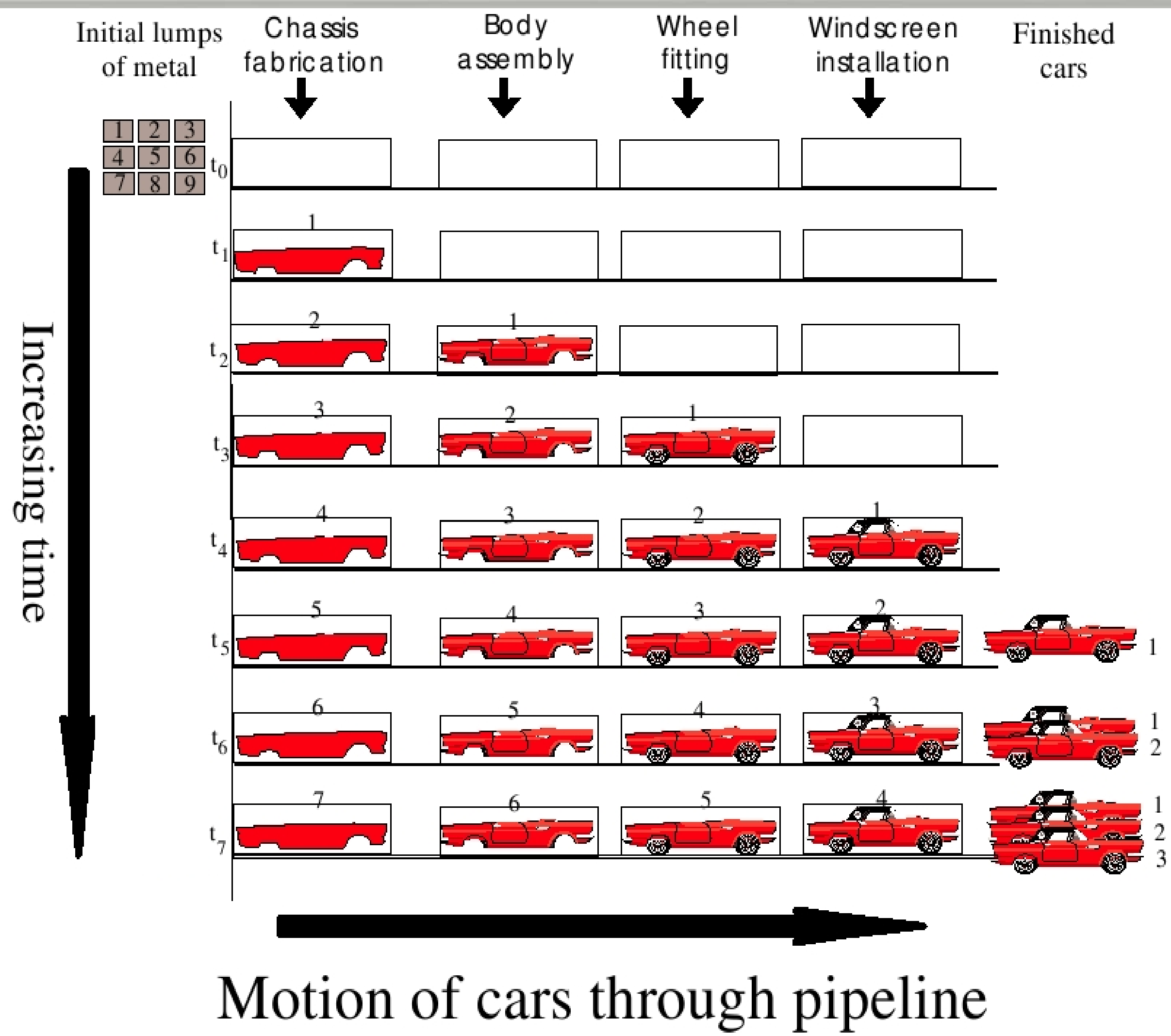
NOTE!

Still need processes to feed the queue and to bleed the queue

Example: Chain Buffer via Replication and Nesting with **PAR**

```
...
  void inputProcess(chanend c){ //FEEDING buffer pipeline
    while (1)
      c <: readPort(); //read input data item, e.g. key from keyboard, sensor data etc...
  }
...
  void outputProcess(chanend c){ //DRAINING buffer pipeline
    int x;
    while (1) {
      c :> x;          //drain element from last buffer element
      printf("%d",x);  //display on screen
    }
  }
...
  void bufferProcess(chanend cIn, chanend cOut){ //BUFFER one element
    int x;
    while (1) {
      cIn :> x;   //try to read (and hold, i.e. buffer)
      cOut <: x;  //try send data item on to next buffer
    }
  }
...
  int main(void) {  //SETUP CONCURRENT PROGRAM
    chan queue[31]; //create 31 channels
    par {          //align 30 concurrently running processes in pipeline
      on stdcore[2]: inputProcess(queue[0]);    //start first process of pipeline
      on stdcore[3]: outputProcess(queue[30]);  //start last process of pipeline
      par (int i=0;i<30;i++) {         //replicate 30 buffer processes
        on stdcore[i%4]: bufferProcess(queue[i],queue[i+1]);
      }
    }
    return 0;
  }
  ...
```

Pipelines: Car making analogy

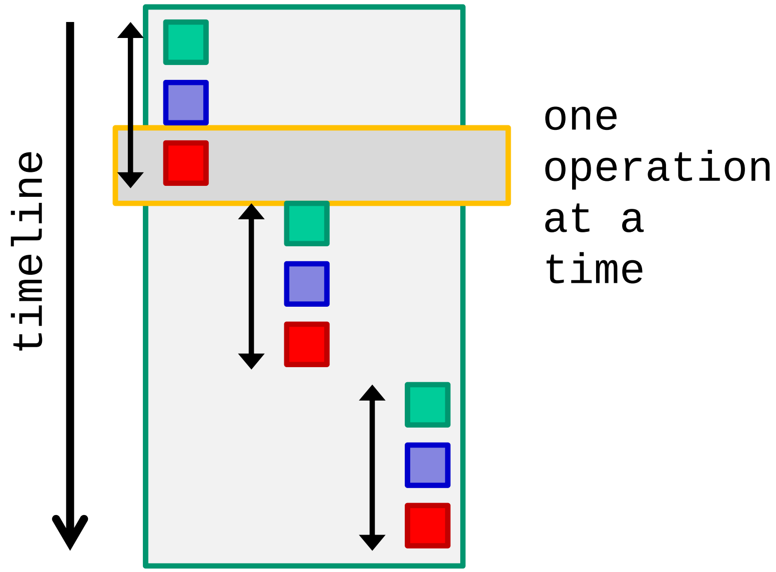


From "Alan Chalmers, Practical Parallel Processing, 1996"

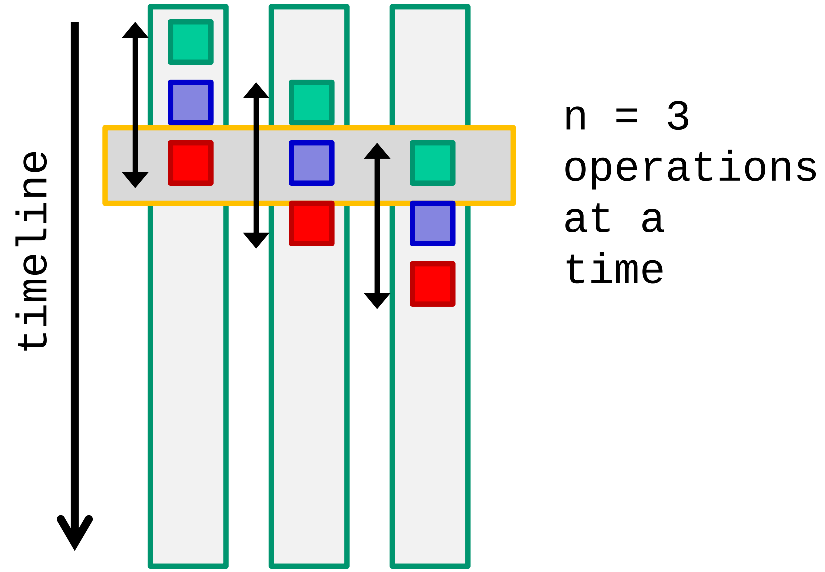
Pipeline Processing

- Pipeline stages transform the data before passing it on in a chain. Can be (much) faster than using a loop.
- Speed advantage of pipeline is due to overlapping in time

Sequential Loop:



Pipeline:



Max time-saving achieved if each stage runs on its own processor.

Pipeline Processing: speed-up?

P1	Op1	Op2	Op3	Op4	Op1	Op2	Op3	Op4	…

Processes 2 pieces of data in 8 time steps

What is the Speed-up?

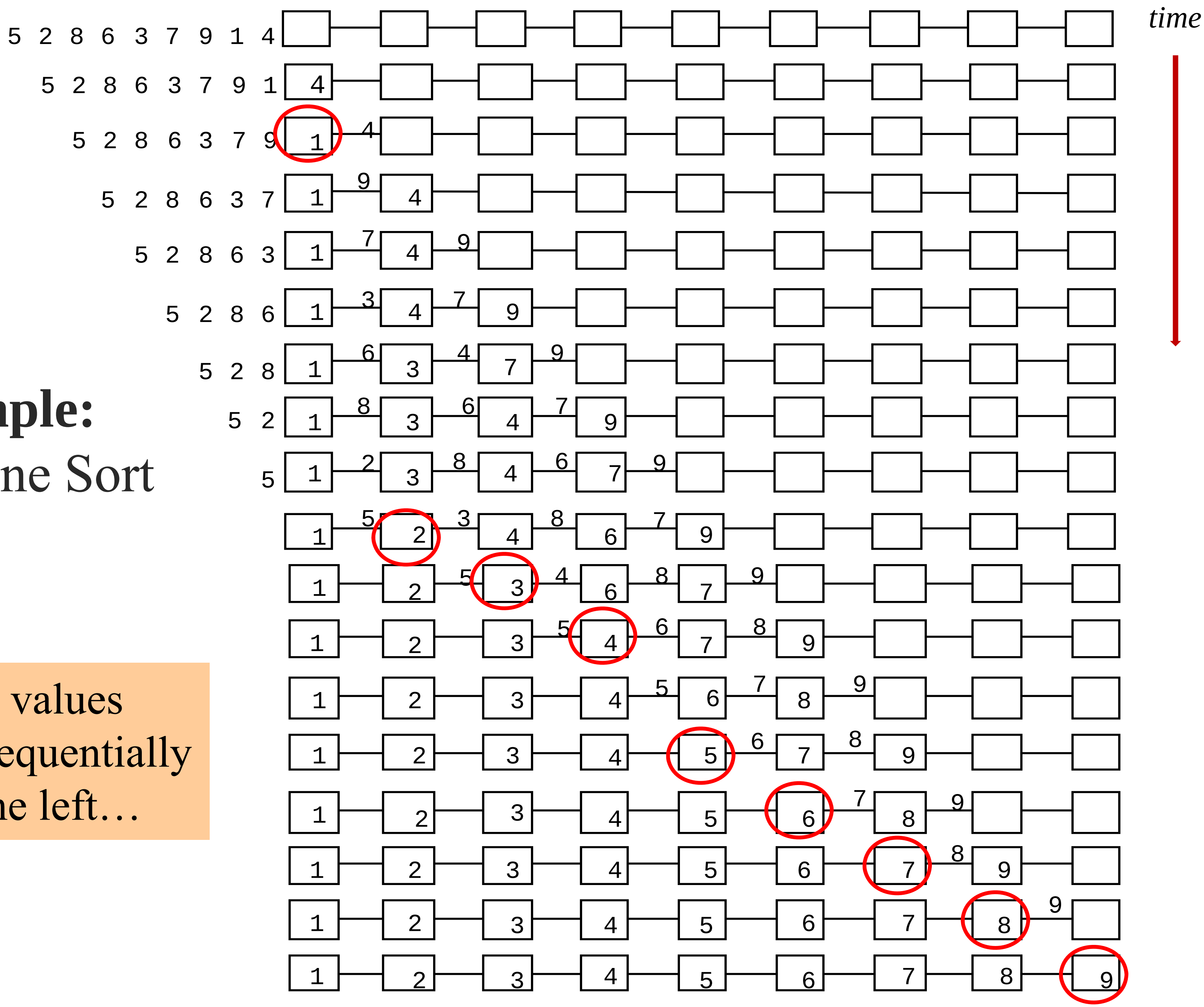
P1	Op1	Op1	Op1	Op1	…				
P2		Op2	Op2	Op2	Op2	…			
P3			Op3	Op3	Op3	Op3	…		
P4				Op4	Op4	Op4	Op4	…	

Processes 4 pieces of data in 7 time steps

Note, we needed 4 processors to get only 2x speed-up.

More on this later…

Example:
Pipeline Sort



…with values input sequentially from the left…

Example: Pipeline Sort

Sort a stream of `N` numbers with a pipeline of `N` processes!

- Each process has two local variables: **`lowest`** and **`next`**.

- Each number enters a process and is compared to value in **`lowest`**.
  - If not smaller than **`lowest`** then pass to next pipeline stage
  - If smaller than **`lowest`** then keep it and pass on previous lowest

- When all numbers are processed, pass final value of **`lowest`** on

→ Develop code using Replicated **`PAR`**!

XC Pipeline Sort Code I

- To sort a sequence of 100 numbers:

```
// XC code to compare numbers for each pipeline stage
...
  int next;
  cIn :> next;
  if (next >= lowest) {
      cOut <: next;
    }
  else {
    cOut <: lowest;
    lowest = next;
  }
...
```

XC Pipeline Sort Code II

- Now repeat comparison code 99 times in each pipeline stage:

```
// XC code for one single pipeline sort stage

void sortStage(chanend cIn, chanend cOut) {
  int lowest;
  cIn :> lowest;
  for (int j=0;j<99;j++) {
    int next;
    cIn :> next;
    if (next >= lowest) {
      cOut <: next;
    }
    else {
      cOut <: lowest;
      lowest = next;
    }
  }
  cOut <: lowest;
}
```

XC Pipeline Sort Code III

```
// XC code fragment for pipeline sort chain
...
void sortStage(chanend cIn, chanend cOut) {
  int lowest;
  cIn :> lowest;
  for (int j=0;j<99;j++) {
    int next;
    cIn :> next;
    if (next >= lowest) {
      cOut <: next;
    }
    else {
      cOut <: lowest;
      lowest = next;
    }
  }
  cOut <: lowest;
}
...
int main(void) {
  chan pipe[101];
  par (int i=0; i<100; i++) {
    on stdcore[i%4]: sortStage(pipe[i],pipe[i+1]);
  }
  return 0;
}
```

- Now replicate 100 times to get 100 parallel processes, one for each pipeline stage
- Note, we need one channel more than the number of processes.

```
chan pipe[101];
```

Main program

XC Pipeline Sort Code IV

- Processes to feed and bleed the pipeline

```
// XC code to feed and bleed pipeline
...
void inputProcess(chanend c) {
  for (int j=0;j<100;j++)
    c <: readUnsortedNumber(j);
}
...
void outputProcess(chanend c) {
  for (int j=0;j<100;j++) {
    int x;
    c :> x;
    printf("%d,",x);
  }
}
...
```

Full XC Pipeline Sort Code

```
void inputProcess(chanend c) {  //FEEDING pipeline
  for (int j=0;j<100;j++)
    c <: readUnsortedNumber(j);
}
...
void outputProcess(chanend c) { //BLEEDING pipeline
  for (int j=0;j<100;j++) {
    int x;
    c :> x;
    printf("%d,",x);
} }

void sortStage(chanend cIn, chanend cOut) { //one STAGE of pipeline
  int lowest;
  cIn :> lowest;
  for (int j=0;j<99;j++) {
    int next;
    cIn :> next;
    if (next >= lowest) {
      cOut <: next;
    }
    else {
      cOut <: lowest;
      lowest = next;
  } }
  cOut <: lowest;
}
...
int main(void) {//SETUP CONCURRENT PROGRAM
  chan pipe[101];
  par (int i=0; i<100; i++) {
    on stdcore[i%4]: sortStage(pipe[i],pipe[i+1]);
  }
  return 0;
}
```

More Efficient **XC Pipeline Sort Code**

```
void sortStage(int i, chanend cIn, chanend cOut) {
  int lowest;
  cIn :> lowest;
  for (int j=0;j<99-i;j++) { //sort unsorted part
    int next;
    cIn :> next;
    if (next >= lowest) {
      cOut <: next;
    }
    else {
      cOut <: lowest;
      lowest = next;
    }
  }
  cOut <: lowest;
  for (int j=0;j<i;j++) { //copy already sorted part
    cIn :> lowest;
    cOut <: lowest;
  }
}
...
int main(void) {
  chan pipe[101];
  par (int i=0; i<100; i++) {
    on stdcore[i%4]: sortStage(i,pipe[i],pipe[i+1]);
  }
  return 0;
}
```

TAKE HOME EXERCISE

- Data already sorted by previous pipeline nodes

→ **do not sort in these cases, but simply copy data**

Parallelism suitable for Pipelining

In general, a program that would otherwise be written using two or more nested **`WHILE`** loops can often be transformed into a pipelined program.

```
WHILE  test1
  WHILE  test2
            ...
```

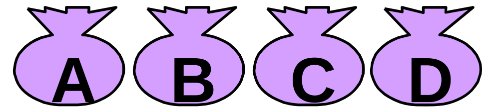
No speed advantage unless the inner loop produces a value to pass on early in its execution.

> if value is only produced at termination, then there is no overlap, hence no parallelism then to be exploited in a pipeline.

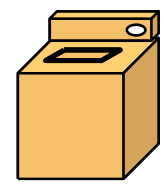
Simple Puzzle: Traditional Pipeline Concept

Dirty Laundry

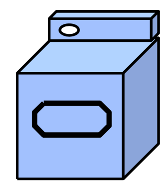
Ann, Brian, Cathy, Dave
each have one load of clothes
to wash, dry, and fold



- Washer takes 30 minutes

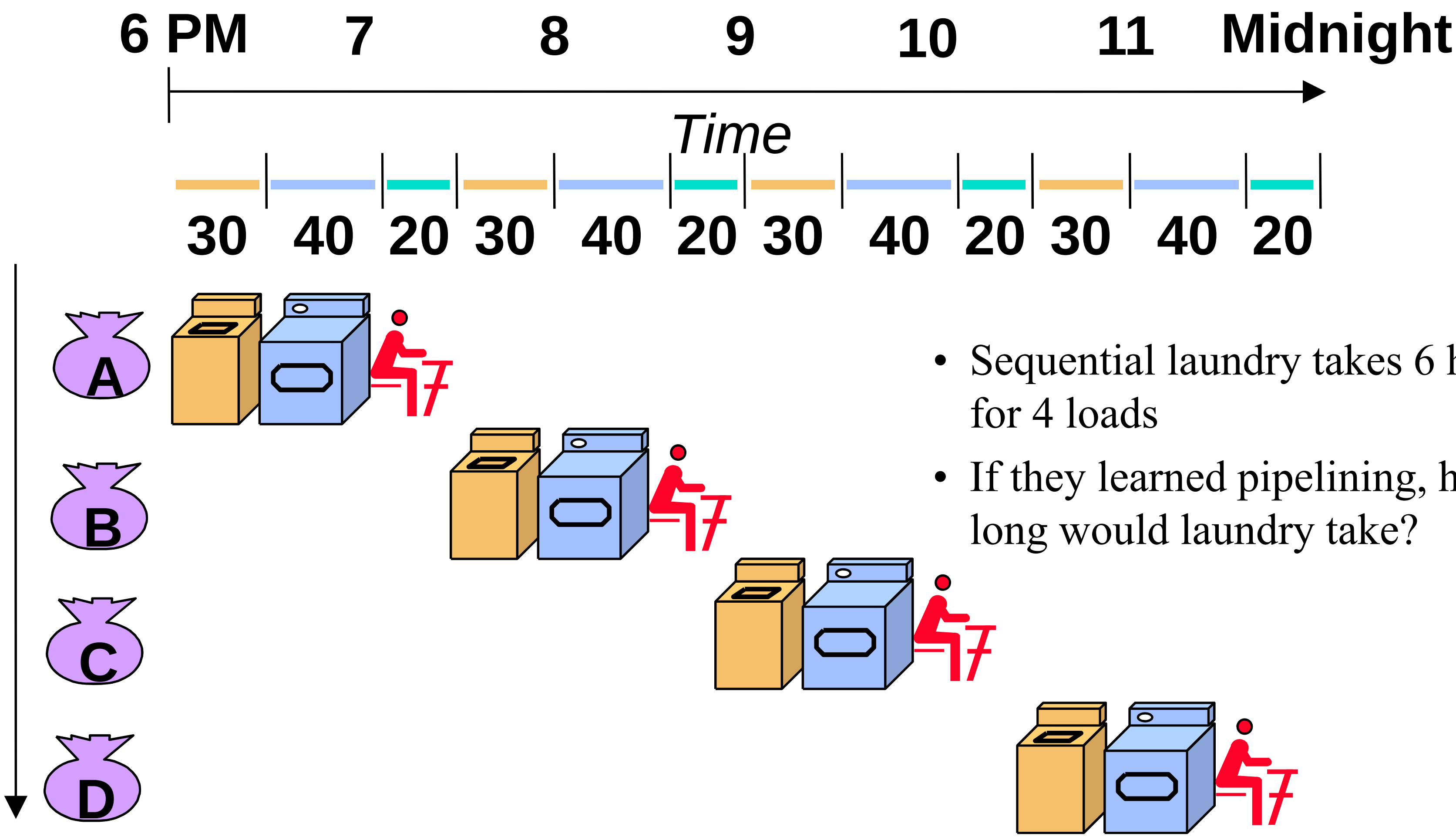


- Dryer takes 40 minutes



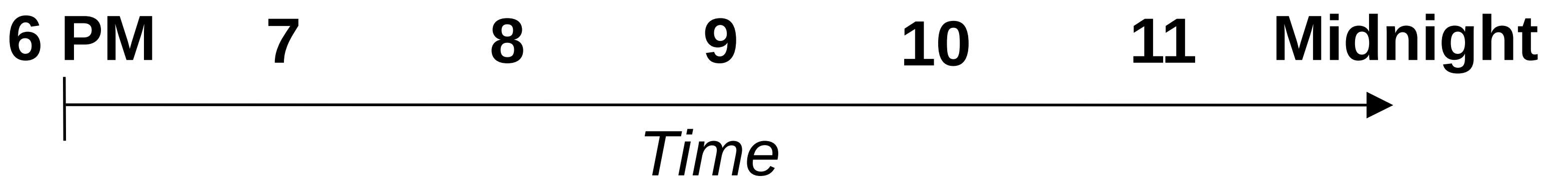
- “Folder” takes 20 minutes

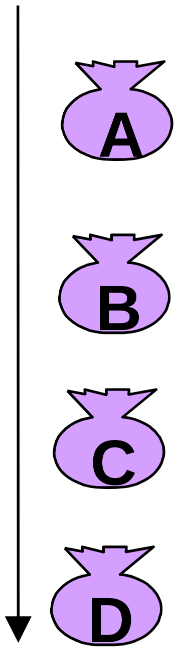
Sequential Approach



- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Simple Pipeline Approach





If they learned pipelining, how long would laundry take and so what time would they finish?

Livelock

A form of deadlock:

when two or more processes continue to execute, but make no progress toward the ultimate goal.

Examples:

- When there is an endless loop in program execution. It occurs when a process repeats itself, because it continues to receive erroneous information.
- When a process that calls another process is itself called by that process, and there is no logic to detect this situation and stop the operation. A livelock differs from a "deadlock," in that processing continues to take place, rather than just waiting in an idle loop.

Analogy

Consider two people walking in a hallway towards each other. The hallway is wide enough for two people to pass. Of interest is what happens when the two people meet. If on the same side of the hallway, a polite strategy is to step to the other side. A more belligerent strategy is to wait for the other person to move. Two belligerent people will suffer in deadlock, glaring face to face in front of each other. Two polite people could suffer from livelock if they repeatedly side-step simultaneously.

Livelock: Basic XC examples


```
{
  while (1)
    ... Do some computation
  ch1 :> x    -- never executed
}
```

Obvious example: an endless loop, and the comms never takes place.

```
while (1) {
  ch1 :> x
  ... Do something
  ch2 <: y
}
```

```
while (1) (
  ... Do something
  ch1 <: x
  ch2 :> y
}
```

Two processes that will infinitely just communicate amongst themselves.

XC: Thread Disjointness Rules [for Variables]

The rules for disjointness on a set of threads $T_0 \ldots T_i$ and *a* set of variables $V_0 \ldots V_j$:

- If thread T_x contains any modification to variable V_m then none of the other threads (T_t ; $t \neq x$) are allowed to use V_m
- If thread T_x contains a reference to variable V_m then none of the other threads (T_t ; $t \neq x$) are allowed to modify V_m
- If thread T_x contains a reference to port V_p then none of the other threads are allowed to use V_p

- a group of threads can have **shared read-only access** to a local variable, but only a single thread can have **exclusive read-write access – IN THEORY, NOT IN XC THOUGH!**

- disjointness rules for variables guarantee that each thread has a well-defined meaning that is independent of the order in which instructions in other threads are scheduled

Examples

```
main (void) {
  int i=1,j=2,k=3;
  par {
    i = k + 1;        // Thread X
    j = k - 1;        // Thread Y
  }
}
```

Legal or not?

```
main (void) {
  int i=1,j=2,k;
  par {
    i = j + 1;        // Thread X
    k = i - 1;        // Thread Y
  }
}
```

Legal or not?

```
main (void) {
  int a[2];
  par {
    a[0] = f(0);      // Thread X
    a[1] = f(1);      // Thread Y
  }
}
```

Legal or not?