Week 9: Scheduling and Concurrency

CS211: Programming and Operating Systems

Niall Madden (Niall.Madden@NUIGalway.ie)

Wednesday and Thursday, 14+15 April 2021

CS211 Assessment

Grades for CS211 will be based on

1. Four programming assignments: 40%. The final component, Lab 6, is due [??DISCUSS??] Suggest 5pm, Monday 3rd May?
2. Homework assignment 20%. Details are on Blackboard. Deadline is 5pm, Friday 30 April. [??DISCUSS??]
3. Online exam: 40%.

~~This will be confirmed in the next few weeks.~~

~~*End of Week 1, Part 1*~~

CS211, this week, will be all about . . .

1. Part 1: Scheduling - examples
   - Algorithms
   - Round Robin (RR)
   - Example
2. Part 2: Concurrency
   - Race condition
3. Part 3: Critical sections
   - Atomic Operations
   - Locks

Wednesday

4. Part 4: Semaphores
   - Previewing Lab 6
   - Coding a semaphore
5. Part 5: Deadlock and Starvation
6. Part 6: Resource Allocation Graphs
   - Example 1
   - Example 2
7. Part 7: The Dining Philosophers
   - Deadlock handling
8. Exercises

Thursday.

Part 1: Scheduling - examples

CS211
Week 9: Scheduling and Concurrency

Start of ...

PART 1: Part 1: Scheduling Processes - examples

Part 1: Scheduling - examples

***For more, see Chapter 7 of the Textbook*.**

Last week we started studying ***SCHEDULING***: algorithms by which the Operating System decides which of the available processes will be given access to the CPU, i.e., set **running**.

First: recall the **states of a process** and how they relate to each other.

Part 1: Scheduling - examples

We studied four **Scheduling Algorithms**:

1. First-Come-First-Served (FCFS) ✓
2. **Shortest-Job-First** (SJF)
3. Shortest Time-to-Completion First (STCF) ✓
4. Round-Robin (RR)

} Preemptive.

For each of these, we consider a few examples of process mix; for each example we'll assumed that processes have a single CPU burst, measured in seconds (though this unit is not important).

Several Processes (aka Jobs). have been submitted to the ready queue. In which order will they be run?

Part 1: Scheduling - examples Scheduling metrics

We compared algorithms according to the following **METRICS**:

1. Turnaround time – the time that elapses between when process arrives in the system, and when it finally completes.
2. Wait time – the amount of time between when a process arrives, and when it completes, that it spends doing nothing.
3. Response time – the time that elapses between when process arrives in the system, and when it executes for the first time.

Part 1: Scheduling - examples

Round Robin (RR)

Each process gets a small unit of CPU time called a ***time quantum*** or ***time slice*** –usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units. (~~***Why?***~~)

The size of the *quantum* is of central importance to the **RR** algorithm. If it is too large, then its is just the FCFS model. If it is too low, them too much time is spent on context switching.

That is, the longest response time is at most $(n-1)q$ time units.

Also, average response time is [see Exercise!]

Part 1: Scheduling - examples

Example

Suppose the following arrive in the following order:

Proc	Arrive Time	Burst Time
P_1	0 ✓	20 ✓
P_2	2 ✓	15 ✓
P_3	4 ✓	10 ✓
P_4	6 ✓	5 ✓

Calculate the

(a) ***Average Turnaround Time***,

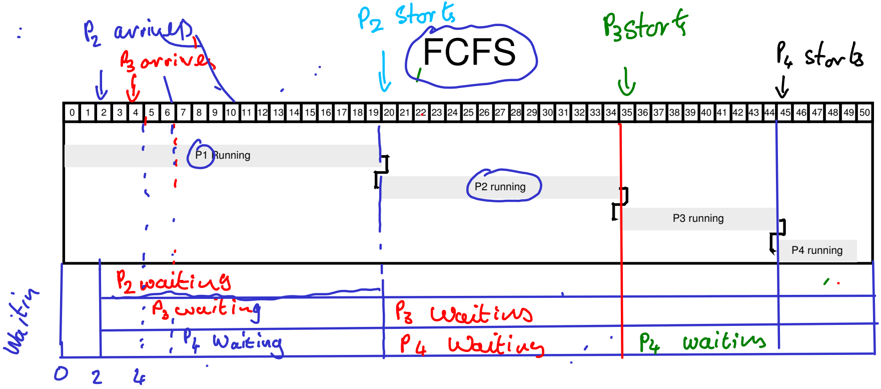
(b) ***Average Wait Time***, and

(c) ***Average Response Time*** for

1. FCFS
2. SJF
3. STCF
4. RR with $q = 10$

Part 1: Scheduling - examples

Example



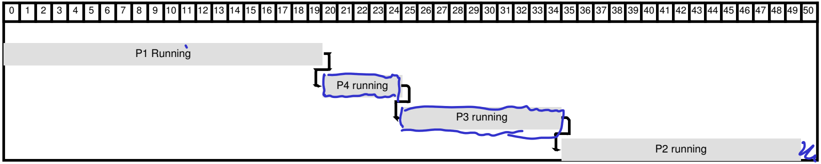
Turnaround times : 20 + 33 + 41 + 44 ,

Average Turnaround is (138/4) = 32.5 second.

Average Wait : (0 + 18 + 31 + 39)/4 = 22.

Response time = Wait time, in this case.

Shortest Job First (SJF)



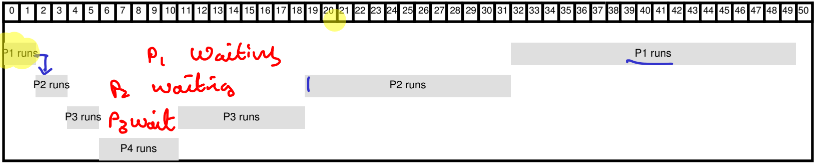
Turnaround time: 20, 48, 31, 19

Average is (118)/4 = 29.5

Wait times: 0, 33, 21, 16 . Average is 70/4 = 17.5

Again Response = Wait.

Shortest Time to Completion First (STCF)



Average Turnaround time is $\frac{1}{4}(50 + 30 + 15 + 5) = \frac{100}{4}$

$= 25$

Response time: $0 + 0 + 0 + 0 = 0$.

Average Wait time: not same as response.

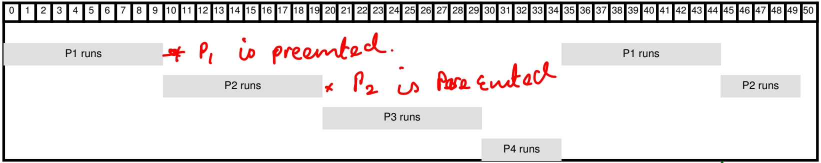
Work out yourself!

Part 1: Scheduling - examples

P_2 arrives

P_2 Runs

Round Robin with $q = 10$



* P_1 is preemted.

* P_2 is Preemted

P_3 finish

P_4 finish

P_1 finish

Check:

Ave Turnaround time is 37?

Ave Resp. time is 15.?

CS211
Week 9: Scheduling and Concurrency

END OF PART 1

Part 2: Concurrency

CS211
Week 9: Scheduling and Concurrency

Start of ...

PART 2: Concurrency

doing more than 1 thing
at a time.

Part 2: Concurrency

Please read Chapter 26 (Concurrency) of the textbook for much more detail on threads.

A *cooperating* process is one that can affect or be affected by another process that is executing on the system.

Threads are prime examples of this: we can think of them as a single process with multiple points of execution. They share program code and, crucially, data.

In this section, we consider the problems that occur then one or more threads try to access the same data, and we look at potential solutions.

A classic data inconsistency problem is the so-called

order of "finishing" matters.

"***Race Condition***",

which we'll study in Lab 6 (a more complicated version is discussed in Sections 26.3-26.4).

Part 2: Concurrency

Race condition

A **Race Condition** (also called a **data race**) is one where the result depends on the order in which instructions are executed.

For a single-thread process, this is predetermined.

But for multi-threaded processes, we do not have control over the order in which individual threads execute their instructions.

Part 2: Concurrency

Race condition

Consider the following example: two cooperating process called P_1 and P_2 share the variable `count`. At various times during execution either may increment or decrement `count`.

count++

The machine usually implements an **increment** as follows:

1. load the value of `count` into a register: $\overset{5}{\underline{REG_1}} = \overset{5}{\texttt{count}}$
2. add 1 to the contents of the register: $\underset{6.}{REG_1} = \underset{5}{REG_1} + \underset{1}{1}$
3. overwrite the contents of `count` with the contents of the register: $\overset{6}{\texttt{count}} = REG_1$.

count- -

A **decrement** would be implemented as

1. load the value of `count` into a register: $\overset{6}{REG_2} = \overset{6}{\texttt{count}}$
2. subtract 1 from the contents of the reg: $\underset{5}{REG_2} = \underset{6}{REG_2} - \underset{-1}{1}$
3. save the contents of the register as `count`: $\underset{5}{\texttt{count}} = REG_2$

Part 2: Concurrency

Race condition

Suppose the value of `count` is 5. If P_1 executes an increment and P_2 executes a decrement, then the value of count should still be 5. Unless the individual operations happen in the following order...

P_1 executes	REG_1	=	$count$	$REG_1 = 5$	✓
P_1 executes	REG_1	=	$REG_1 + 1$	$REG_1 = 6$	✓
P_2 executes	REG_2	=	$count$	$REG_2 = 5$	✓
P_2 executes	REG_2	=	$REG_2 - 1$	$REG_2 = 4$	✓
P_1 executes	$count$	=	REG_1	$count = 6$	
P_2 executes	$count$	=	REG_2	$count = 4$	

We arrive at the wrong state because we allowed both threads to manipulate the variable `count` at the same time.

Since the outcome depends on the order in which each operation takes place, we have a ***race condition***.

CS211
Week 9: Scheduling and Concurrency

END OF PART 2

CS211
Week 9: Scheduling and Concurrency

Start of ...

PART 3: Critical sections

Short summary

→ Read these notes in your own time.

Part 3: Critical sections

Critical Section

(From Section 26.4 of the text-book). A **critical section** is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.

The example given on Slides 18 and 19 shows that multiple threads executing the same code can result in a race condition, that is an example of a **critical section.**

To resolve this, we would like to enforce **mutual exclusion**: This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

Part 3: Critical sections

Atomic Operations

One possible solution is to make the operation **"atomic"** (or ***indivisible***. This is, the critical section is executed as though it were a single operation, and so impossible to interrupt.

In a realistic setting, that is not possible for all race conditions. But, as we will see, the use of some atomic operations can help us solve the larger problem, by creating **locks**.

Part 3: Critical sections Locks

See Section 28.1 of the textbook

So now we know we would like to execute a series of instructions atomically. But, in general, on a multiprocessor system, we can't.

But what we can do is create a **lock** which we put around critical sections, and thus ensure that any such critical section executes as if it were a single atomic instruction.

Part 3: Critical sections

For the **lock** approach to work, the following 3 conditions must be satisfied:

- **Mutual Exclusion:** If process T_i is executing in its critical section, then no other processes can be executing in their critical sections.
- **Fairness/Progress:** If there are some procs that wish to enter the critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely.
- **Performance/Bounded Waiting:** after a process has made a request to enter its critical section and before that request is granted, then must be a **bound** (i.e., a limit) on the number of times other processes are allowed to enter their critical sections.

Part 3: Critical sections

Synchronization Hardware

We consider two basic approaches to this:

1. Interrupt suspension
2. Automatic ***test-and-set()*** and ***swap()*** instructions.

1. Interrupt suspension (Section 28.5)

Suppose a process is in its critical section. If it cannot be preempted then data consistency should be maintained. On a single processor system, the problem could be solved by disabling interrupts while a shared variable is begin modified. However, such a method is not feasible on a multiproc system: large over-heads would be incurred informing all procs that interrupts are dis-allowed.

Read yourself

Part 3: Critical sections Synchronization Hardware

2 Automatic instructions

The processor has facilities to swap the contents of two words (in memory), or test and change the contents of a word *automatically* – i.e., as a single instruction.

There are other hardware and software solutions to synchronization problems. The most important, perhaps, is a tool known as a ***semaphore*** (Chapter 31).

Read this

CS211
Week 9: Scheduling and Concurrency

END OF PART 3

Finished here Wednesday.

Part 4: Semaphores

CS211
Week 9: Scheduling and Concurrency

Start of ...

PART 4: Semaphores

Part 4: Semaphores = "Sign".

A Semaphore S is an integer variable that can only be accessed via one of two operations: often, binary - either 0 or 1.

(Test/*sem_wait*) $P(S)$:

```
while (S ⩽ 0)
      { wait(); }
  S--;
```

Requesting to enter the critical section

not available

available.

implement all this as a read from a pipe.

(Increment/*sem_post*) $V(S)$:

```
S++;
```

Release the resource.

(Historically, these functions were called Probern (Dutch for "test") and Verhogen (increment)).

These operations must be ***indivisible*** (or "atomic"). This is, when one process (or thread) modifies a semaphore value, no other process can modify it at the same time.

Part 4: Semaphores

There are two types of semaphore:

Most Important!

1. **Binary semaphores (locks):** These are used to control access to a single resource, such as a memory location. If the resources is available then $S = 1$. Otherwise $S = 0$. When a process wants to access it,
   - (i) it calls the function $P(S)$
   - (ii) enters it critical section
   - (iii) calls $V(S)$ when it exits the critical section.
2. **General (or counting) semaphores:** These are used to control access to a pool consisting of a finite number of identical resources. Say there are 5 units available. The S is initialised to 5. Whenever a process requests the resource, it calls $P(S)$ and decrements the value of S. If S reaches 0 then the next proc that requests that resource must wait until another frees it by running $V(S)$.

Part 4: Semaphores

Previewing Lab 6

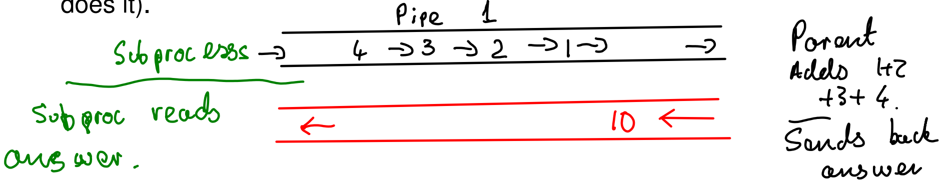
Lab 6 will be all about coding a senario when a **race condition** can occur, and then solving it.

pipe : first-in-first-out queue

For consider the example in `adder.c`.

- a (parent) process creates a subprocess (i.e., "child process").
- Then the subprocess tries to calculate the sum of 4 numbers by placing them in a pipe, from where the parent will read them.
- The parent sums the four numbers, and places the result in another pipe for the child to read.
- The child reads this solution and prints it.

Since there are no competing processes, nothing should go wrong (not does it).



Part 4: Semaphores

Version 1: no problem

adder.c (main)

```
int main(void )
{
  int ParentsPID, ans;
  pipe(inpipe);
  pipe(outpipe);
  ParentsPID = getpid(); // now I'll always know who I am
  fork(); // Now have 2 procs. Subprocess will have diffent pid
  if ( getpid() == ParentsPID ) ✓
    Adder();   // The parent will be the adder
  else
  {
    ans = SubProc(1,2,3,4);
    printf("SubProc (%d): 1+2+3+4= %d\n", getpid(), ans);
  }
  return(0);
}
```

for send data from subPROC to parent, and back.

child.

Part 4: Semaphores

Version 1: no problem

adder.c: adder(), run by parent

```
void Adder(void ) // run by parent
{
  int i, number, sum=0;

  for (i=0; i<4; i++)
  {
    read(inpipe[0], &number, sizeof(int));
    sum += number;
  }
  write(outpipe[1], &sum, sizeof(int));
}
```

| Reads 4 numbers from pipe.

Part 4: Semaphores

Version 1: no problem

adder.c: SubProc(), run by subprocess (“child”)

```c
int SubProc(int a, int b, int c, int d)
{
  int ans;
  printf("SubProc (%d) writes four numbers to the pipe()\n",
         getpid());
  write(inpipe[1], &a, sizeof(int));
  write(inpipe[1], &b, sizeof(int));
  sleep(1); // Pause for a second to encourage race condition
  write(inpipe[1], &c, sizeof(int));
  write(inpipe[1], &d, sizeof(int));

  printf("SubProc (%d) reads the answer from a pipe()\n",
         getpid());
  read(outpipe[0], &ans, sizeof(int));
  return(ans);
}
```

Part 4: Semaphores

Version 1: no problem

The output I get when I run this is:

```
SubProc (4285) writes four numbers to the pipe()
SubProc (4285) reads the answer from a pipe()
SubProc (4285): 1+2+3+4= 10
```

So - no problem!

Part 4: Semaphores

Version 2: problem!

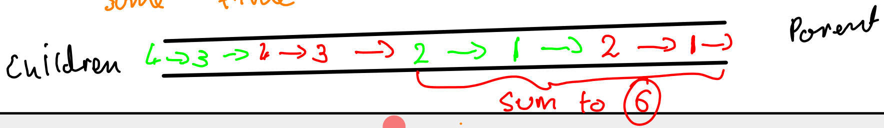
But in the next version, the parent has two children both doing the same thing. See adder_race_condition.c

Now the output is

```
SubProc (4485) writes four numbers to the pipe()
SubProc (4486) writes four numbers to the pipe()
SubProc (4485) reads the answer from a pipe()
SubProc (4485): 1+2+3+4= 6
SubProc (4486) reads the answer from a pipe()
SubProc (4486): 1+2+3+4= 14
```

In Lab 6 we'll design a semaphore solution to this problem

Since these were writing to the pipe at some time



Part 4: Semaphores

Coding a semaphore

Recall that a **Semaphore** S is an integer variable that can only be accessed via one of two operations: **Test/*sem_wait*** $P(S)$, and **Increment/*sem_post*** $V(S)$.

A more detailed explanation of semaphores

See slide 30

Implementing semaphores with `pipe()` in C.

Idea: represent a semaphore, S, as a pipe. If S=0, pipe is empty. If S=1, there is one item in the pipe.

So Test (P) read one value from pipe. This works because, if the pipe is Empty, the reading process waits.

To Increment (V): write to the pipe,

CS211
Week 9: Scheduling and Concurrency

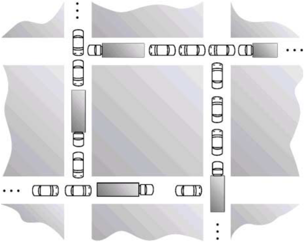
END OF PART 4

Part 5: Deadlock and Starvation

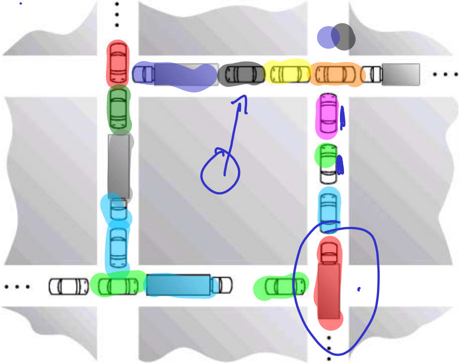
CS211
Week 9: Scheduling and Concurrency

Start of ...

PART 5: Deadlock and Starvation



Part 5: Deadlock and Starvation



Deadlock is when two or more procs are waiting indefinitely for an event that can only be caused by one of the waiting processes. E.g., all are stuck in the `wait()` loop of the $P()$ function.

Part 5: Deadlock and Starvation

Deadlock can arise if four conditions hold simultaneously

1. ***Mutual exclusion***: only one process can have access to a particular resource at any given time. ✓
   → can't move until ~~we~~ there is space ahead available
2. ***Hold and wait***: a process holding at least one resource is waiting to acquire additional resources held by other processes.
3. ***No preemption***: a resource can only be released voluntarily by the process holding it, after that process has completed its task. → no crane!
4. ***Circular wait***: there exists a set $\{P_0, P_1, \ldots, P_n, P_0\}$ of waiting processes such that
   - P_0 is waiting for a resource that is held by P_1,
   - P_1 is waiting for a resource that is held by P_2, ...,
   - P_{n-1}, is waiting for a resource that is held by P_n which in turn is waiting for P_0.

only one car can occupy a space at a time.

Part 5: Deadlock and Starvation

CS211
Week 9: Scheduling and Concurrency

END OF PART 5

Part 6: Resource Allocation Graphs

CS211
Week 9: Scheduling and Concurrency

Start of ...

PART 6: Resource Allocation Graphs

Part 6: Resource Allocation Graphs

Deadlocks may be described in using a directed graph called a ***resource allocation graph***.

graph where edges have arrows.

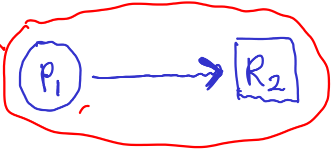
This graph has two sets of **vertices:**

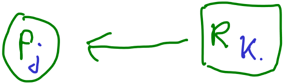
- **Processes:** $P_0, P_1, \ldots, P_n$ and
- **Resources:** $R_0, R_1, \ldots, R_m$

and ***Edges***

- from P_j to R_k if process j as requested resource k but not yet been allocated it,
- from R_k to P_k j if process j as been allocated resource k and not yet released it.

Eg





Part 6: Resource Allocation Graphs Example 1

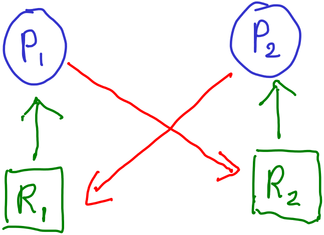
Example (Example 1)

A system has $m = 2$ resources, R_1 and R_2, and $n = 2$ processes, P_1 and P_2.

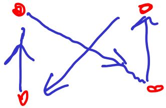
- P_1 has been allocated R_1, and is requesting R_2.
- P_2 has been allocated R_2, and is requesting R_1.

Draw the resource allocation graph for the scenario. Does the system reach deadlock?

Cycle is sequence of neighbouring vertices that starts & end at some vertex.



This is deadlocked.



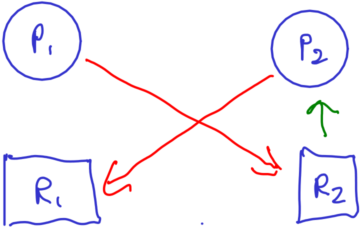
Part 6: Resource Allocation Graphs

Example (Example 2)

A system has $m = 2$ resources, R_1 and R_2, and $n = 2$ processes, P_1 and P_2.

- P_1 is requesting R_2.
- P_2 has been allocated R_2, and is requesting R_1.

Draw the resource allocation graph for the scenario.
Does the system reach deadlock?



No cycle — not dead-locked

Eg. P_2 can have both R_1 & R_2. when it is finished, P_1 gets R_2.

Part 6: Resource Allocation Graphs — Summary

- If there are no cycles, there is no deadlock,
- If there is deadlock, there must be a cycle
- If there is a cycle, there ***may*** be deadlock
- If each resource has only one instance, and there is a cycle, then there is deadlock

CS211
Week 9: Scheduling and Concurrency

END OF PART 6

Part 7: The Dining Philosophers

CS211
Week 9: Scheduling and Concurrency

Start of ...

PART 7: The Dining Philosophers Problem

Part 7: The Dining Philosophers

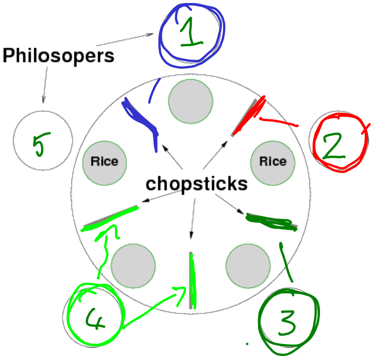
(See also Section 31.6 of the textbook)

Starvation occurs when a particular process is always waiting for a particular semaphore to become available.

The ideas of **Deadlock** and **Starvation** are exhibited in the classic synchronization problem: ***The Dining Philosophers Problem***.

- There are five philosophers seated at a round table.
- Each has a bowl of rice in front of them and a chopstick to their left and right.
- They spend their day alternating between eating and thinking.
- However there are only five chopsticks...

Part 7: The Dining Philosophers



If a philosopher is hungry, she will try to pick up the chopstick to the left and then the chopstick to the right. If she manages to do this they will eat for a while before putting down both and thinking for a while. However, if she it picks up one, she will not let go of it until she can picks up the second and eat.

Suppose each of the picks up the fork to their left. No forks remain on the table so we reach a state of deadlock.

The challenge is to find a solution so that

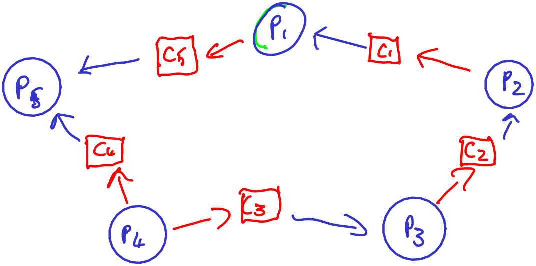
- **Deadlock** does not occur.
- neither does **starvation** where one philosopher never gets to eat.

Part 7: The Dining Philosophers

Example (Taken from 1718-CS211 Exam)

In the dining philosophers problem, each philosopher wants to pick up the 2 forks beside him/her so that they can eat. Suppose we have 5 such philosophers and

- at time $t = 0$: nobody has picked up any fork
- at time $t = 1$: Philosophers 1, 2 and 3 have picked up the forks to their left, philosopher 4 has not picked up any fork and philosopher 5 has picked up two forks.



Part 7: The Dining Philosophers Deadlock handling

In general operating systems take one of three approaches to deal with deadlock:

1. Ensure that the system will never enter a deadlock state.
2. Allow the system to enter a deadlock state and then recover.

In Case 1, there are two possibilities:

- ***Prevention:*** we ensure at least one of the four necessary conditions never hold.
- ***Deadlock avoidance:*** where the OS uses ***a priori*** information about the procs the devise an algorithm to circumvent deadlock.

In Case 2, the OS must have mechanisms for first detecting deadlock and then dealing with it.

Finished here Thursday

Exercises

Exercise (9.1)

(This is taken from the CS211 Semester 2 from 2017/2018)

Given the data (all time in seconds)

Process	*Arrival time*	*Process duration*
P1	*3*	*5*
P2	*1*	*3*
P3	*0*	*8*
P4	*4*	*6*

for four processes,

determine the scheduling result for the policies of

1. *Round Robin (with time quantum 4)*
2. *First Come First Served*

(c) Calculate the average turnaround time and average waiting time for these examples.

Note: The “wait time” of a process is the length of time it spends doing nothing.

Exercises

Exercise (9.2)

Draw the resource allocation graph for the scenario where all philosophers pick up the fork/chopstick to their left.

Exercise (9.3)

A system has $m = 4$ identical resources, and $n = 3$ processes, P_1, P_2 and P_3, which make a request for 1, 2, and 3 resources, respectively. Draw the resource allocation graph for the scenario. Can the system reach deadlock?