OpSys: Oblig #1

120912 – Victor Rudolfsson

Contents

Chapter 1: Theory

1.6.1 Forklar kort hva som er de to hovedoppgavene til ett operativsystem

The two main functions for an operating system are 1) taking care of basic hardware operationgs – controlling input/output and detecting hardware failure for example; and 2) to manage and interact with software.

1.6.2 Hva het det operativsystemet som anses som forløperen till UNIX og når omtrent ble UNIX lansert første gang? Hvem skrev den første utgaven av UNIX?

UNIX was first released in the late 1960s or early 1970s as a sort of lite successor to MULTICS. The first version of UNIX was authored by Ken Thompson.

1.6.3 Hva mener vi med begrepet *multiprogrammering* og hvilke fordeler innebærer dette I forhold til ren batch-kjøring?

Multiprogramming refers to a way of multitasking, by allowing several programs to run at the "same time". In contrast to 'batch-kjøring', where programs are queued up and run in order, with multiprogramming several instances can seem to run at the same time on the same processor by swiftly switching between them. By first executing part of the first program and then switching to the other program, this creates an illusion of having multiple programs executing at the same time.

Chapter 2: Theory

2.12.1 Hva er ett *Directive* i assemblykode?

A directive is an instruction to the assembler (in contrast to an instruction run by the CPU), that tells the assembler something it may need to know to assemble the program. A directive commonly begins with a dot.

2.12.2 Hva forbinder du med begrepene *superskalar* og *pipelining* i en prosessor?

Pipelining is the process of queueing up work to be done by the CPU (as if they were put in a pipeline); and a superscalar CPU is a CPU that can do different types of calculations simultaneously (integer calculation, float. calculation, etc).

2.12.3 Hva menes med *kernel* og *user mode?*

Kernel mode and user modes are differently privileged modes programs may run in. For example, processes in *user mode* are not allowed to do I/O, but must instead call for the kernel to perform those operations and return back to the process in user mode.
When running in kernel mode, there are no restrictions on what memory addresses may be accessed or what instructions may be run, unlike when running in user mode.

2.12.4 Hva menes med *hyperthreading/multithreading* som ble innført med Pentium 4?

Hyperthreading is Intel's proprietary name for what is otherwise known as multithreading, which is a method of parallelization that creates virtual cores for each CPU core, and allows the CPU to run one thread on each virtual core and simulate running these at the same time.

2.12.5 Hva er ulempen ved en direkte mappet ("direct mapped") cache?

The high risk of hash collisions.

2.12.6 Anta en CPU som skal sjekke om dataene på en gitt adresse i minne finnes i en sett-assosiativ cache. Hvordan finner CPU'en frem til riktig sett og riktig vei ("way")?

First, the CPU locates the appropriate set by its index, then scans it to decide whether it's a cache hit or miss.

2.12.7 Nevn tre måter interrupt kan oppstå på i en datamaskin.

1. Internal interrupt: This kind may arise from problems in execution, or erronous operations
2. External Interrupts: This type may occur i.e when an I/O-device interrupts what
3. Software interrupts: This type occurs when we trap to the kernel, i.e through system calls

2.12.8 Hvor store cache lines har en 2-veis sett-assosiativ cache som er 32KB stor og har 128 sett?

x*2*128 = 32KB
x = 32KB/256b = 128b

Chapter 3: Theory

3.6.1 Hva menes med *preemptive* og *non-preemptive* operativsystemer?

In a preemptive operatingsystem, the kernel can be interrupted by I.e system calls when doing other things, and switch between processes; whereas in a non-preemptive operatingsystem a process is guaranteed to run until it exits. In real-time systems, for example, we may know very well that processes will only take a very short amount of time to do their job and it makes sense for [some] them to be non-preemptive.

3.6.2 Hva mener vi når vi sier at et OS er et *monolittisk system*?

In a monolithic design, the operating system is contained in *one* binary file, where the kernel alone runs in kernel mode. Windows, Linux, Minux, etc are all monolithic. BUT NOW I'M NOT SURE ANYMORE BECAUSE APPARENTLY ALL OF THEM (AND OS X TOO) ARE HYBRIDS.

3.6.3 Hva mener vi når vi sier at en trend innen OS-design har gått I retning av *micro-kernel*? Hvilke fordeler gir ett slik OS-design?

I would presume this means there's an ongoing trend of wanting a hybrid between a monolithic and a microkernel design, because of the flexibility and stability provided by the microkernel design (where the kernel is as small as possible and most functionality is provided by modules or services) and the speed provided by a monolithic design (because of a lower amount of context switches?).

3.6.4 Forklar sammenhengen mellom systemkall, kommandoer og instruksjoner. Gi eksempler.

A system call is a call to the kernel to do something which the process running in user land does not have access to do. A command could involve a system call but does not necessarily have to, whereas both of these often consist of multiple instructions to the CPU.

Doing I/O requires system calls, so the ***command(s)*** “echo $((1+1)) > file” would amongst others give the CPU an instruction to calculate 1+1, cause a system call *write* to write the returned value to the file specified.

3.6.5 Beskriv mest mulig detaljert hva som skjer når systemkallet read(fd,&buffer,nbytes) utføres.

The kernel is told to *read* a certain amount of bytes as specified by *nbytes* from the *file descriptor (fd)* which specifies what file and where it should read from. Upon finishing, the results should be returned to the calling process.

3.6.6 Hvilke utskrifter kan vi få når denne koden kjøres om vi forutsetter at fork() systemkallet ikke feiler?

Either 0 (in the child process) or p where p is the process ID of the spawned child.

Chapter 3: Lab

3.7.2

```
01      Code starts here
02      Global function main
03      Declaration of main
04              push base pointer on stack ( { )
05              set stack pointer to base pointer
06              define variable as 0, push on stack
07              compare integer value 4 bytes down the stack with 0
08              if not equal go to section .L2
09              increment value 4 bytes down the stack by 1
10      Here begins section .L2
11              store 0 in register
12              gtfo
13
```

In C:

```
int main()
{
        int x = 0;
        if(x == 0)
                x++;
        return 0;
}
```

3.7.3

```
01      Code starts here
02      Global function main
03      Here starts main
04      {
05              set stack pointer to base pointer
06              store value 0 on stack
07              unconditional jump to while at line 11
08      Here begins section L3
09              Add value 1 to value found 4 bytes down from current position of base pointer
10              Add value 1 to value found 4 bytes down from current position of base pointer
11      Here begins  section L2
12              compare value 9 with value found 4 bytes down from current position of base pointer
13              jump (if comparison less than or equal is true) to section .L3
14      store 0 in register
15      gtfo
16
```

In C:

```
int main()
{
  int y = 0;
  while(y <= 9)
  {
    y++;
    y++;
  }
  return 0;
}
```

Chapter 4: Theory

4.4.1 Hva er en prosess i et operativsystem?

A process is an instance of a program.

4.4.2 Beskriv tre viktige tilstander en prosess kan befinne seg i.

A process can be in ready state (waiting to run), blocked state (temporarily done, or interrupted), or running state (currently running).

4.4.3 Hva er en prosesskontrollblokk (PCB) og hva inneholder den?

A process control block contains all information needed to properly keep track of a process when switching between multiple processes. It's a data package containing information such as the process' ID number, register content, pointers to the topmost and bottommost part of its memory space, list of files opened by the process, states of flags, and the status of I/O devices the process needs.

4.4.4 Lag en liste over egenskaper som deles av tråder i en prosess og egenskaper som er unike for hver tråd.

Threads share:

- Heap memory
- Memory address
- Page tables
- Process state
- I/O status information
- Signals
- Environment variables

Threads do not share:

- Registers
- Stack
- Per-thread data

4.4.5.Hva skjer om vi får et blokkerende I/O-kall i en tråd på user level?

The thread should then block and wait for the I/O call to complete. Meanwhile, it allows another waiting thread to use the CPU.

4.4.6 Forklar hvilke (og hvor mange) trådtabeller som finnes når:

- tråder kjøres på user level?

With user level threads, each process has a table of all its threads.

- tråder kjøres på kernel level?

With kernel level threads, the kernel manages all the threads and thus keeps them all in one table.

Chapter 4: Labs

4.5.1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

/**
**  I interpreted this assignment not as *starting*
**  the processes at the given times but as the processes
**  performing their work at these times; because I felt
**  I couldn't wait to start process 1 in the parent
**  until process 0 was done, but start process 2
**  before it was.
**
**  After attempting to write it with a for loop, and then
**  recursively I realized waitpid() only worked on *children*
**  and not on siblings.
**
**  The reason for attempting that was because I made the assumption
**  that the reason for giving the processes numbers were
**  that they were supposed to be started in that order,
**  and that they should all be started from a common
**  parent process.
**
**  In this code, I have instead resorted to not starting the threads
**  in numerical order but in the order that makes sense, and
**  giving them the appropriate number. All code is on the following page
**  to make reading easier.
**
**
**/
```

```
void process(int number, int time)
{
    printf("Prosess %d kjører\n", number);
    sleep(time);
    printf("Prosess %d kjørte i %d sekunder\n", number, time);
}

int main()
{
    pid_t p[6];

    p[0] = fork();

    if(p[0] != 0)
    {
        // Start 2 at the same time
        p[2] = fork();

        if(p[2] != 0)
        {

            // Start 1 after 0
            waitpid(p[0], NULL, 0);
            p[1] = fork();

            if(p[1] != 0)
            {
                // Start 4 at the same time as 1
                p[4] = fork();

                if(p[4] != 0)
                {
                    if(waitpid(p[2], NULL, 0) &&
                    waitpid(p[1], NULL, 0))
                    // Start 3 when 2 and 1 finishes
                    p[3] = fork();

                    if(p[3] != 0)
                    {

                        // Start 5 when 4 finishes
                        waitpid(p[4], NULL, 0);
                        p[5] = fork();

                        if(p[5] != 0)
                            waitpid(-1, NULL, 0); // Wait for all children
                        else
                            process(5, 3);
                    }
                    else
                        process(3, 3);

                }
                else
                    process(4, 2);
            }
            else
                process(1, 3);
        }
        else
        {
            waitpid(p[0], NULL, 0);
            process(2, 2);
        }

    }
    else
        process(0, 1);

    exit(0);
}
```

4.5.2

First, this code will spawn a child through forking, after which the child will call the print-loop (and then exit), whereas the parent will also call the print-loop, wait for the child to exit, and then exit itself.

With both the child and the parent in the write-loop, they will loop and increment a loop variable *i,* and every 100 000 iteration, *g_ant* will be first incremented, and then the value of it will be printed together with a text indicating whether this is the child or the parent doing this. The global variable g_ant will be the same for both the child and the parent as they both start out, but after forking the child will only hold a copy of the variable, meaning they both increase different instances of “the same” variable.

Once the loop has iterated 3 000 000 times, or rather, g_ant has reached 30 (for each process), the write loop finishes and child calls exit(0) whereas the parent makes sure the child has quit before it returns 0 and exits.

4.5.3

This code is very similar to 4.5.2 but with a vital difference: instead of creating a child *process*, it creates a child *thread*. This means that they share global variables not by value but by reference, and thus modifying the value of it in one thread will also change it in the other. When they enter the write-loop, they will both try to increment the same variable, and so this program will print half the amount of lines since it's not two variables of the same value but one variable shared between them that will be incremented, as a result of sharing the same memory space.

Chapter 5: Theory

5.5.1 I læreboka figur 2.21 finnes et eksempel med to prosesser som ønsker utskrift. Hvilken prosess sørger for å hente utskriftfilene fra spooler directory? Hvorfor kan det gå galt med denne løsningen?

Because the actions are not atomic. If one process attempts to get the value of the variable (like how many items are in the queue, or a pointer to the end of the queue) and before it has updated it, the next process attempt to get the value of the same variable, both now operate with the same value. If both of them then updates that value, one of these updates will be lost when the other one overwrites it.

5.5.2 Når brukes busy waiting for å synkronisere prosesser?

Busy waiting is used to continuously query the state of a variable, for example to make sure a shared resource is only accessed by one process, or waiting for another process to terminate as is the case with waitpid(). This is also called a spinlock.

5.5.3 De fleste CPUer har en assemblyinstruksjon som heter TSL eller XCHG. Forklar hvordan en av disse virker, og hva som er gjør at den er nyttig ved synkronisering av prosesser.

When accessing shared data, using a flag to indicate whether this data is being accessed by another process may feel appropriate. However, this flag is shared data itself, and so a race condition can occur on the flag, and we're back on square one. However, the TSL, Test-and-Set-Lock is an atomic instruction built into the CPU, which allows the CPU to both query the state of the flag and set it in one instruction. This makes it especially useful for synchronizing processes to avoid race conditions – because it can all be done in one instruction without opening up for a race condition to the flag.

5.5.4 Forklar hva wait- og signal-operasjonene i ovenst aende programlinjer har som funksjon. Hva er hensikten med variabelen mutex og hvorfor forekommer den i Wait-operasjonen?

The wait operation blocks on the variable cond, and releases the mutex. The signal operation unblocks all other threads waiting for the conditional variable cond. The mutex is used for accessing the conditional variable.

5.5.5 Forklar hvorfor utskriften fra dette programmet ikke nødvendigvis blir slik vi ønsker at den skal bli

Because threads are not being synchronized, and there is no lock on the variable, we do not have full control of the modification of g_ant.

I would set up a mutex, and set a mutex-lock in the critical region of the while-loop, row *05*, and release the mutex only after it has been printed, row *07*.

Chapter 5: Labs

5.6.1

```
#include <stdio.h>     /* printf */
#include <stdlib.h>    /* exit */
#include <unistd.h>    /* fork */
#include <pthread.h>
#include <semaphore.h>
#define SHARED 1

sem_t happy_semaphore[6];

struct threadargs
{
  int id;         // Thread's ID
  int sleeptime;  // Sleep this long
  int signal[6];  // When done, signal thread(s) with ID=key and value=1
};


/* thread function: start by waiting on my own semaphore if it has been
 * initialized to 0; do my work (sleeping); signal the threads that
 * should start when I finish; exit
*/
void* tfunc(void *arg)
{
  struct threadargs *arguments=arg;
  int i;

  sem_wait(&happy_semaphore[arguments->id]);

  printf("Tråd %d kjører\n", arguments->id);
  sleep(arguments->sleeptime);
  printf("Tråd %d er ferdig og vekker kanskje andre...\n", arguments->id);

  for(i = 0; i < 6; i++)
      if(arguments->signal[i] == 1)
        sem_post(&happy_semaphore[i]);
}

int main(void)
{
  int i,j;
  pthread_t tid[6];
  struct threadargs *thread_arguments[6];

  /* allocate memory for threadargs and zero out semaphore signals */
  for(i=0;i<6;i++)
  {
    thread_arguments[i] = (struct threadargs*) malloc(sizeof(struct threadargs));
    for(j=0;j<6;j++) thread_arguments[i]->signal[j]=0;
  }

  /**
   ** Define sleeptime, what thread(s) to wake
   ** and ID for each thread.
   */
  thread_arguments[0]->id=0;
  thread_arguments[0]->sleeptime=1;
  thread_arguments[0]->signal[1]=1;
  thread_arguments[0]->signal[4]=1;

  thread_arguments[1]->id=1;
  thread_arguments[1]->sleeptime=2;
  thread_arguments[1]->signal[3]=1;

  thread_arguments[2]->id=2;
  thread_arguments[2]->sleeptime=3;
```

```
  thread_arguments[3]->id=3;
  thread_arguments[3]->sleeptime=2;


  thread_arguments[4]->id=4;
  thread_arguments[4]->sleeptime=3;
  thread_arguments[4]->signal[5]=1;

  thread_arguments[5]->id=5;
  thread_arguments[5]->sleeptime=3;


  sem_init(&happy_semaphore[0], SHARED, 1);
  sem_init(&happy_semaphore[1], SHARED, 0);
  sem_init(&happy_semaphore[2], SHARED, 1);
  sem_init(&happy_semaphore[3], SHARED, 0);
  sem_init(&happy_semaphore[4], SHARED, 0);
  sem_init(&happy_semaphore[5], SHARED, 0);

  for(i=0; i < 6; i++)
    pthread_create(&tid[i], NULL, tfunc, (void *) thread_arguments[i]);


  for(i=0;i<6;i++) pthread_join(tid[i], NULL);

  return 0;
}
```

5.6.2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>     /* usleep */
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#define SHARED 0        /* process-sharing if !=0, thread-sharing if =0*/
#define BUF_SIZE 20
#define MAX_MOD 100000
#define NUM_ITER 1000

void *Producer(void *); /* Producer thread */
void *Consumer(void *); /* Consumer thread */

struct arguments
{
    int id;
};

sem_t empty;            /* empty: How many empty buffer slots */
sem_t full;             /* full: How many full buffer slots */
pthread_mutex_t mutex;  /* mutex: Mutex lock */
int data[BUF_SIZE];     /* shared finite buffer  */

int main(int argc, char* argv[])
{
  if(argc < 2)
    printf("\nToo few arguments bro, please provide amount of producers/consumers.");

  int i,j, amount = atoi(argv[1]);


  // Argument (containing only ID) to pass each thread
  struct arguments *thread_arguments[amount];

  for(i=0;i<amount;i++)
  {
    thread_arguments[i] = (struct arguments*) malloc(sizeof(struct arguments));
    thread_arguments[i]->id=i;
  }

  pthread_t pid[amount], cid[amount];

  sem_init(&empty, SHARED, BUF_SIZE);
  sem_init(&full, SHARED, 0);
  pthread_mutex_init(&mutex,0);

  printf("Main started; setting up %d producers and consumers ...\n",amount);

  for(i=0; i<amount; i++)
  {
    pthread_create(&pid[i], NULL, Producer, (void*)thread_arguments[i]);
    pthread_create(&cid[i], NULL, Consumer, (void*)thread_arguments[i]);
  }

  for(i=0; i<amount; i++)
  {
    pthread_join(pid[i], NULL);
    pthread_join(cid[i], NULL);
  }

  printf("main done\n");

  return 0;
}


void *Producer(void *arg)
```

```
{
  struct arguments *args = arg;

  int semvalem,i=0,j;          /* semvalem: semaphore value of empty semaphore */

  while(i < NUM_ITER)
  {
    usleep(rand()%MAX_MOD);    /* pretend to generate an item by a random wait*/
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);
    sem_getvalue(&empty, &semvalem);
    data[BUF_SIZE-(semvalem-1)]=1; /* put item in buffer */
    /* the following two lines just prints a bar showing current buffer fill */
    j=BUF_SIZE; printf("(Producer %d, semaphore empty is %d) \t",args->id,semvalem);
    while(j > semvalem) { j--; printf("="); } printf("\n");
    pthread_mutex_unlock(&mutex);
    sem_post(&full);
    i++;
  }

  return 0;
}


void *Consumer(void *arg)
{
  int semvalfu,i=0,j;       /* semvalfu: semaphore value of full semaphore */
  struct arguments *args = arg;

  while(i < NUM_ITER) {
    usleep(rand()%MAX_MOD); /* pretend to generate an item by a random wait*/
    sem_wait(&full);
    pthread_mutex_lock(&mutex);
    sem_getvalue(&full, &semvalfu);
    data[semvalfu]=0;       /* remove item from buffer */
    /* the following two lines just prints a bar showing current buffer fill */
    j=0; printf("(Consumer %d, semaphore full is %d)  \t",args->id,semvalfu);
    while(j < semvalfu) { j++; printf("="); } printf("\n");
    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
    i++;
  }

  return 0;

}
```

Chapter 6: Theory

6.9.1 Hva mener vi n ar vi sier at schedulingen er non-preemptive? Hvilke systemer bruker preemptive og non-preemptive scheduling?

Preemptive scheduling means processes can be interrupted to allow for another process to run for a bit; whereas non-preemptive scheduling means the process runs until the process is done. Preemptive scheduling is very common in most operating systems we use in our daily lives, whereas real-time systems usually use non-preemptive scheduling, i.e because they know a process will finish relatively shortly and give place for another one.

6.9.2 Hva ligger i begrepet “starvation” ifb med Shortest Job First (SJF) scheduling algoritmen?

When using Shortest-Job-First, as the name implies, the shortest jobs will be run before longer jobs. However, if more jobs are continuously added that are relatively short, longer jobs in the queue may not get a chance to run and thus, they may starve to death.

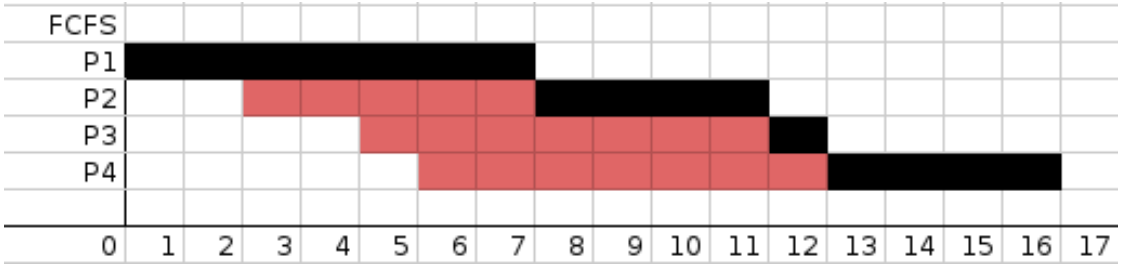
6.9.3 Anta Round Robin scheduling. Vurder konsekvensene ved korte og lange tidskvantum.

When using Round Robin scheduling, each process gets a set amount of time to run before it's interrupted. But changing the process also takes some time, because the process' data has to be brought back into memory/cache, and so on. If this were to take 1ms and we gave each process a 5ms quantum, there would be a 20% time loss for performing this _every_5_millisecond. If we instead allowed a 100ms quantum for each process, we would reduce that percentage, but instead get a slow response to, for example, short interactive requests.

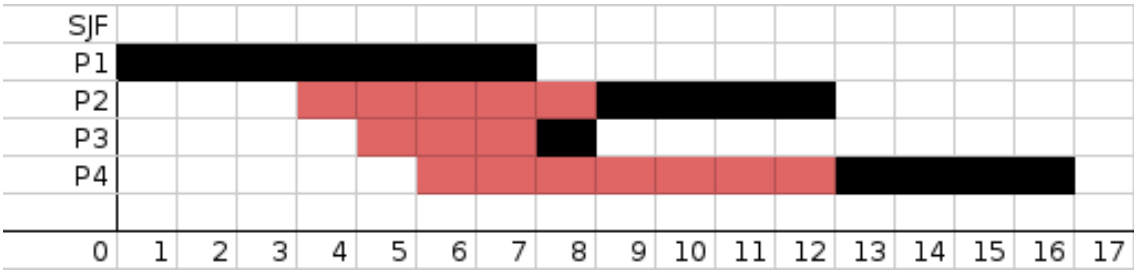
6.9.4 Nevn noen fordeler vi har ved prioritert scheduling.

With weighted or prioritized scheduling, one of the most obvious benefit is that requests that are more important get handled quicker. For example, a user who runs Dropbox in the background syncing files and decides to open his web browser is probably more satisfied if his or her direct requests are prioritized higher than background jobs such as Dropbox syncing. At the same time, the overhead is neither too big or too small.

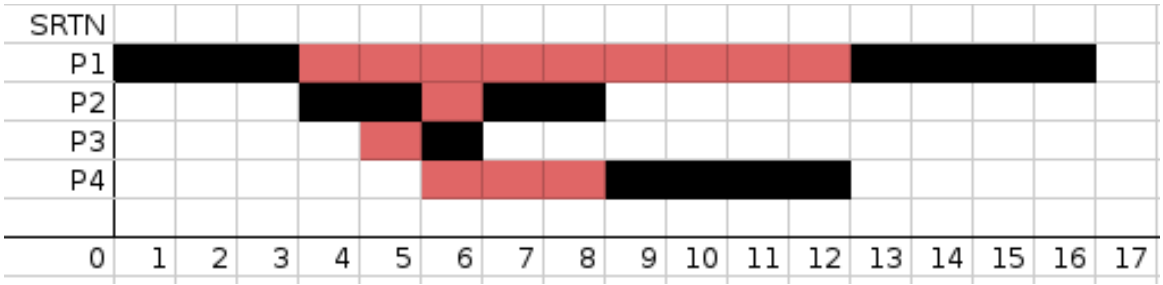
6.9.5 Anta ett sett med fire prosesser med ankomst-tid og burst/CPU-tid (i ms) gitt i tabell. Tegn et Ganttskjema (tidstabell) som illustrerer eksekveringen av prosessene for hver av følgende schedulingsalgoritmer. Hva blir gjennomsnittlig turnaroundtid for hver av schedulingsalgoritmene?



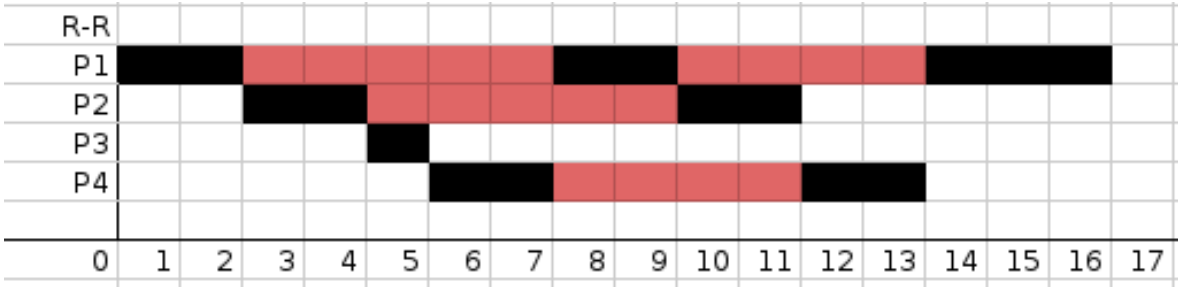
Average turnaround: (7 + 9 + 8 + 11) / 4 = 35/4 = 8.75



Average turnaround: (7+10+4+11) / 4 = 32 / 4 = 8



Average turnaround: (16+6+2+7) / 4 = 31/4 = 7.75



Average turnaround: (16+9+1+8) = 34/4 = 8.5